MICRO FOCUS

Artix 5.6.3

Artix for CORBA C++

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK

http://www.microfocus.com

2015-02-11

Contents

Preface

What is Covered in This Book

This book describes a variety of different CORBA integration scenarios and explains how to use the Artix command-line tools to generate or modify WSDL contracts and IDL interfaces as required. Details of Artix programming, however, do not fall within the scope of this book.

Who Should Read This Book

This book is aimed at engineers already familiar with CORBA technology who need to integrate Web services applications with CORBA.

The Artix Documentation Library

For information on the organization of the Artix library, the document conventions used, and where to find additional resources, see ***Using the Artix Library***, available with the Artix documentation at https://supportline.microfocus.com/productdoc.aspx.

Contacting Micro Focus

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

Further Information and Product Support

Additional technical information or advice is available from several sources.

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

- The WebSync service, where you can download fixes and documentation updates.
- The Knowledge Base, a large collection of product tips and workarounds.
- Examples and Utilities, including demos and additional product documentation.

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

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

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

Information We Need

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

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

Contact information

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

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

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

You may want to check these URLs in particular:

- http://www.microfocus.com/products/corba/artix.aspx (trial software download and Micro Focus Community files)
- https://supportline.microfocus.com/productdoc.aspx (documentation updates and PDFs)

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

http://www.microfocus.com/Resources/Newsletters/infocus/newsletter-subscription.asp

Introduction to CORBA Web Services

Artix provides a flexible framework for bridging between CORBA and Web Services domains. Several different approaches can be used to integrate a CORBA application into a Web Services domain and this introduction provides a brief overview of some typical integration scenarios.

Artix Architecture

The key feature of the Artix architecture is that it supports multiple communication protocols. With the help of the plug-in development APIs, moreover, it is possible to extend Artix to support *any* custom protocol.

Figure 1 illustrates this multi-protocol support, showing an Artix application that is capable of sending or receiving operation invocations over three different protocols: SOAP/MQ, SOAP/HTTP, and IIOP.

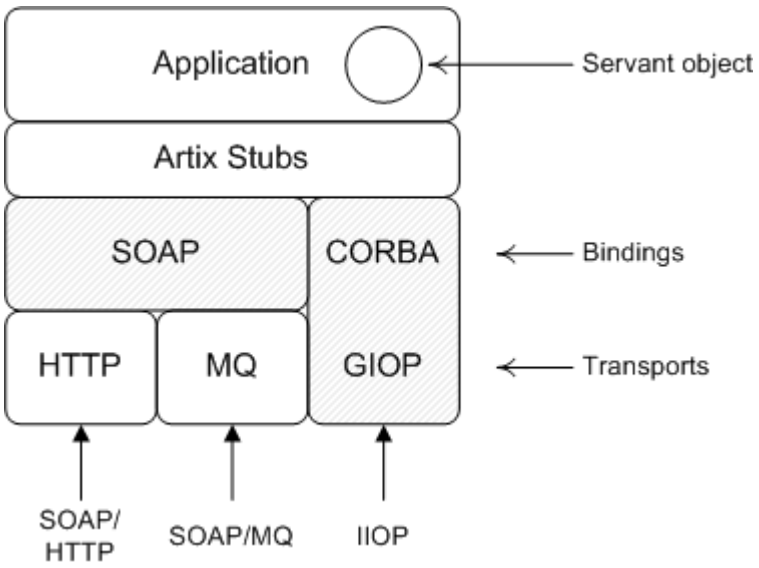

Figure 1: *Artix Application with Multiple Bindings and Transports*

WSDL contract

The Web Services Definition Language (WSDL) contract plays a central role in Artix. It defines the interfaces (or *port types*) and operations for a Web service. In this respect, the WSDL contract is

analogous to an IDL interface in CORBA. However, WSDL contracts contain more than just interface definitions. The main elements of a WSDL contract are as follows:

- *Port types*—a port type is analogous to an IDL interface. It defines remotely callable operations that have parameters and return values.
- *Bindings*—a binding describes how to encode all of the operations and data types associated with a particular port type. A binding is specific to a particular protocol—for example, SOAP or CORBA.
- *Port definitions*—a port contains endpoint data that enables clients to locate and connect to a remote server. For example, a CORBA port might contain stringified IOR data.

Servant object

An Artix servant provides the implementation of a port type (analogously to the way in which an Orbix servant provides the implementation of an IDL interface). The servant class is implemented using the appropriate language mapping (a Micro Focus proprietary mapping for C++).

Artix stubs

The Artix stub contains the code that is needed to encode and decode the messages received and sent by an Artix application. Artix provides command-line tools to generate the stub code from WSDL, as follows:

- `wsdltocpp` command—generates C++ stub code from WSDL..

Bindings

A binding is a particular kind of encoding for operations and data types (for example, CORBA or SOAP). Support for a binding is enabled by loading the relevant plug-in (for example, the `soap` plug-in for SOAP, or the `iiop` plug-in for CORBA, and so on).

In addition to loading the relevant plug-in, you must also provide an XML description of the binding in the WSDL contract. Artix provides tools that will generate the binding for you automatically; there is no need to write them by hand.

Transports

A transport is responsible for sending and receiving messages over a specific transport protocol (for example, HTTP or MQ-Series). Support for a transport is enabled by loading the relevant plug-in (for example, the `mq` plug-in for MQ-Series, or the `at_http` plug-in for HTTP).

In Artix, transports are closely associated with port definitions. For example, if you include either a `<http-conf:client/>` or a `<http-conf:server/>` tag within the scope of a `port` element, this indicates that the port uses the HTTP transport.

Artix routers

An Artix router is used to bridge operation invocations between different communication protocols. Figure 2 shows an example of a SOAP/HTTP-to-CORBA router. This router translates incoming SOAP/HTTP request messages into outgoing IIOP request messages. On the reply cycle, the router translates incoming IIOP reply messages into outgoing SOAP/HTTP reply messages.

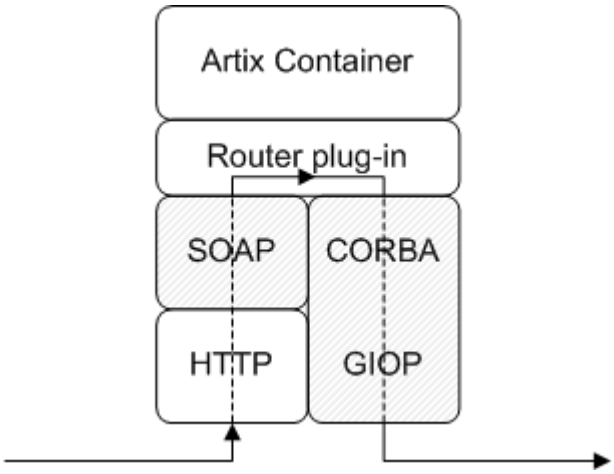

Figure 2: *Example of a SOAP/HTTP-to-CORBA Router*

Artix container

The Artix container, `it_container`, is an application that can be used to run any of the standard Artix services. The functionality of the container is determined by the plug-ins it loads at runtime.

By loading the `router` plug-in (along with the requisite binding and transport plug-ins) the container is configured to run as a standalone router.

Router plug-in

The router plug-in implements a general-purpose protocol bridge. Messages that arrive on one port are sent out on another port.

For example, the router plug-in shown in Figure 2 on page 3 receives request messages over the SOAP/HTTP protocol and forwards the request message out again over the IIOP protocol.

Routes

To configure a router, you need to specify which ports are connected to which other ports. Use the `ns1:route` element to connect a source port to a destination port. For example:

```
<ns1:route name="route_0">
    <ns1:source      service="tns:<SourceService>"
                     port="<SourcePort>"/>
    <ns1:destination service="tns:<DestinationService>"
                     port="<DestinationPort>"/>
</ns1:route>
```

Integrating a CORBA Server with Web Services

This section considers the problem of a legacy CORBA server that is to be opened up to Web services applications. Artix supports a variety of solutions to this integration problem, which are briefly described in the following subsections.

This section contains the following subsections:

- Accessing the CORBA Server through a Standalone Router
- Accessing the CORBA Server through an Embedded Router
- Replacing the WS Client by an Artix Client
- Replacing the CORBA Server by an Artix Server

Accessing the CORBA Server through a Standalone Router

One of the simplest ways to integrate a WS client with a CORBA server is to deploy a *standalone router* to act as a bridge between them. This approach can be used in any system.

Figure 3 shows a CORBA server that is accessible through a standalone router. The router is responsible for mapping incoming SOAP/HTTP requests into outgoing IIOP requests.

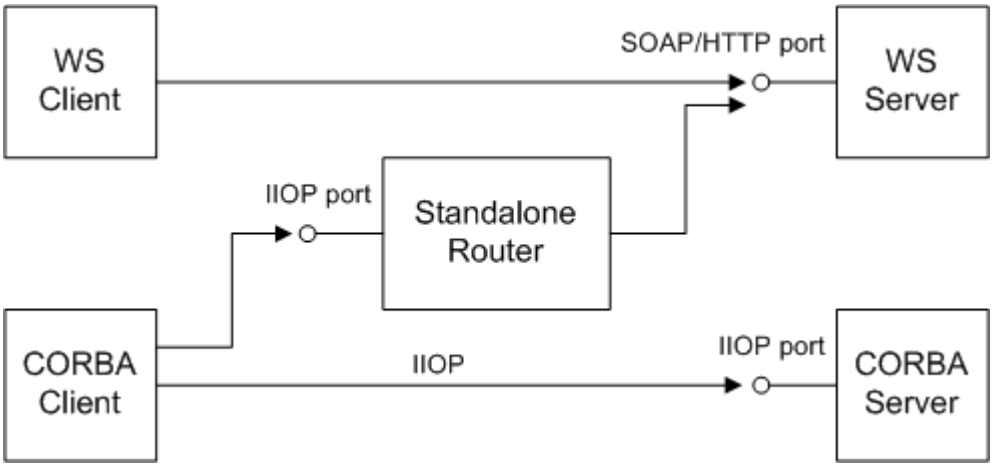

Figure 3: *WS Client Accesses CORBA Server through Standalone Router*

Advantages and disadvantages

This scenario offers the following advantages:

- Compatible with any CORBA server.
- Compatible with any WS client.
- Non-intrusive—no changes need be made either to the client or to the server.

And the following disadvantage:

- Loss of performance—every operation invocation that passes through the router consists of two remote invocations (client-to-router followed by router-to-server).

Accessing the CORBA Server through an Embedded Router

If the CORBA server is implemented using an Orbix 6.x product, it is usually possible to embed the Artix router directly into the Orbix executable. This approach yields significant performance gains.

Figure 4 shows an example of a CORBA server that is accessible through an embedded router. The router is responsible for mapping incoming SOAP/HTTP requests into colocated IIOP requests.

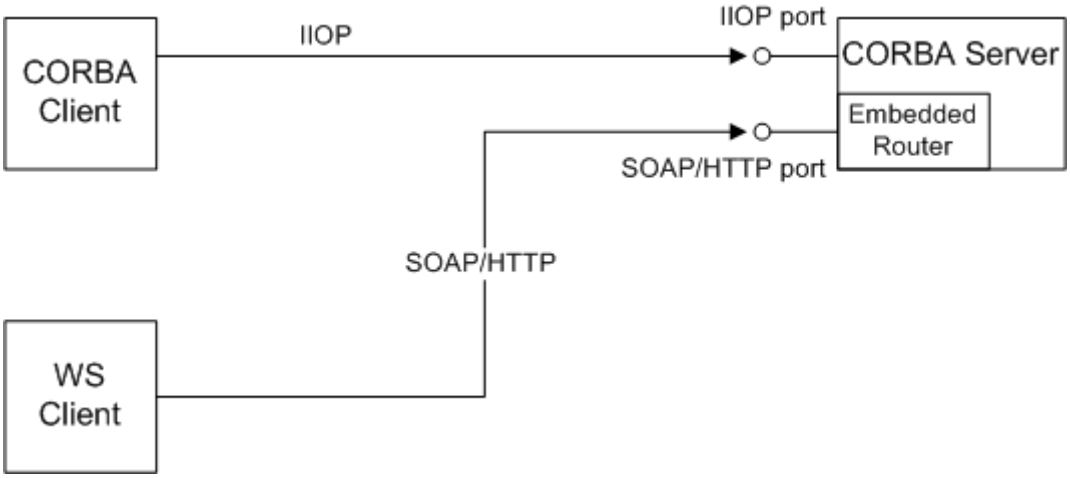

Figure 4: *WS Client Accesses CORBA Server through Embedded Router*

Advantages and disadvantages

This scenario offers the following advantages:

- Compatible with Orbix 6.x implementations of the CORBA server.
- Compatible with any WS client.
- No changes need be made to the WS client.
- The CORBA server must be reconfigured, but remains otherwise unchanged.

And the following disadvantage:

- Moderate performance—this scenario is more efficient than using a standalone router, but is not as efficient as some other scenarios.

Replacing the WS Client by an Artix Client

If you have not implemented the WS client yet, you could implement it using Artix. An Artix client offers great flexibility, because it can communicate through multiple protocols, including IIOP and SOAP/HTTP.

Figure 5 shows an example of a CORBA server that is accessed by an Artix client and a CORBA client. The Artix client is configured to talk directly to the CORBA server using the IIOP protocol.

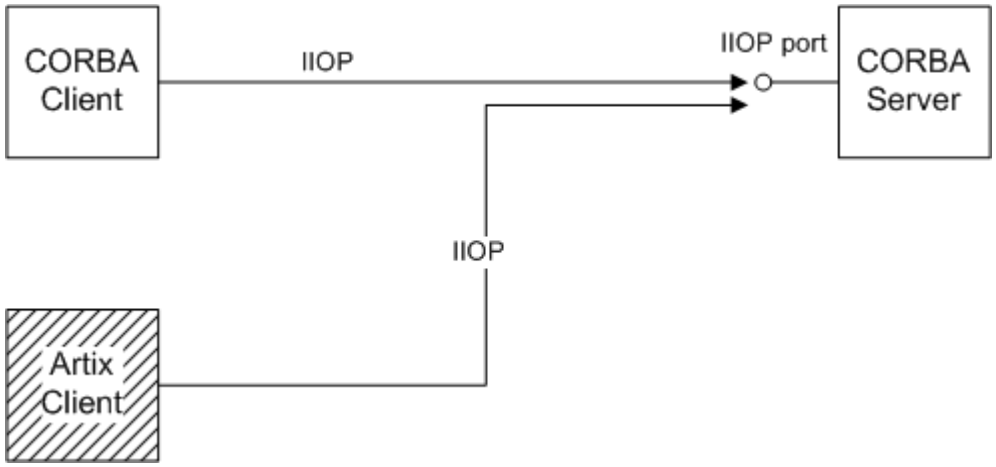

Figure 5: *Replacing the WS Client by an Artix Client*

Advantages and disadvantages

This scenario offers the following advantages:

- Compatible with any CORBA server.
- No changes need be made to the CORBA server.
- Performance is optimized.
- Artix client offers flexibility for future integration.

And the following disadvantage:

- If you have already implemented the WS client, you would have to re-write it to use the Artix APIs.

Replacing the CORBA Server by an Artix Server

If you want to exploit the full power of the Artix product, you might find it worthwhile to replace the CORBA server by re-implementing it as an Artix server. Because Artix supports multiple protocols, an Artix server can easily support present and future integration requirements.

Figure 6 shows an example of an Artix server that is accessed by a WS client and a CORBA client. The Artix server is configured to accept requests both from CORBA clients and WS clients.

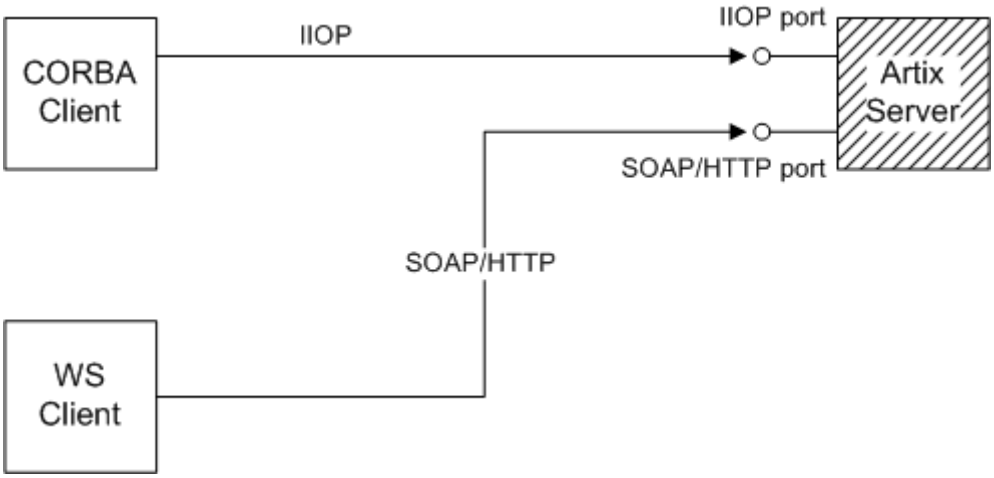

Figure 6: *Replacing the CORBA Server by an Artix Server*

Advantages and disadvantages

This scenario offers the following advantages:

- Compatible with any WS client.
- No changes need be made to the WS client.
- Performance is optimized.
- Artix server offers flexibility for future integration.

And the following disadvantage:

- You must re-implement the CORBA server as an Artix server.

Integrating a CORBA Client with Web Services

This section considers the problem of CORBA client that needs to access a Web services server. Artix supports a variety of solutions to this integration problem, which are briefly described in the following subsections.

This section contains the following subsections:

- Accessing the WS Server through a Standalone Router
- Replacing the CORBA Client by an Artix Client
- Replacing the WS Server by an Artix Server

Accessing the WS Server through a Standalone Router

A relatively simple way to integrate a CORBA client with a WS server is to deploy a *standalone router* to act as a bridge between them. This approach can be used in any system.

Figure 7 shows a WS server that is accessible through a standalone router. The router is responsible for mapping incoming IIOP requests into outgoing SOAP/HTTP requests.

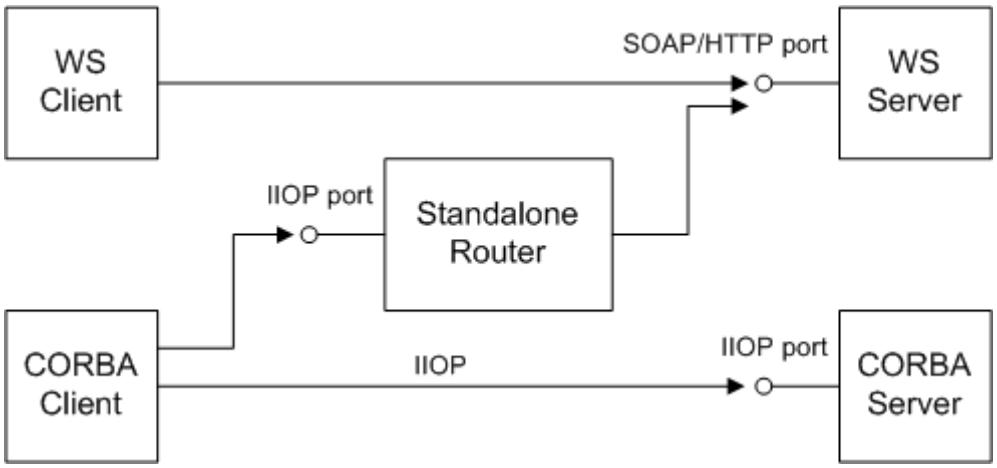

Figure 7: *Client Accesses the WS Server through a Standalone Router*

Advantages and disadvantages

This scenario offers the following advantages:

- Compatible with any WS server.
- Compatible with any CORBA client.
- Non-intrusive—no changes need be made either to the client or to the server.

And the following disadvantage:

- Loss of performance—every operation invocation that passes through the router consists of two remote invocations (client-to-router followed by router-to-server). This has a noticeable impact on performance.

Replacing the CORBA Client by an Artix Client

To exploit the full power of the Artix product, you might find it worthwhile to replace the CORBA client by re-implementing it as an Artix client. The Artix client can then communicate using a wide variety of protocols, including IIOP and SOAP/HTTP.

Figure 8 shows an example of a WS server that is accessed by an Artix client and a WS client. The Artix client is configured to talk directly to the WS server using the SOAP/HTTP protocol.

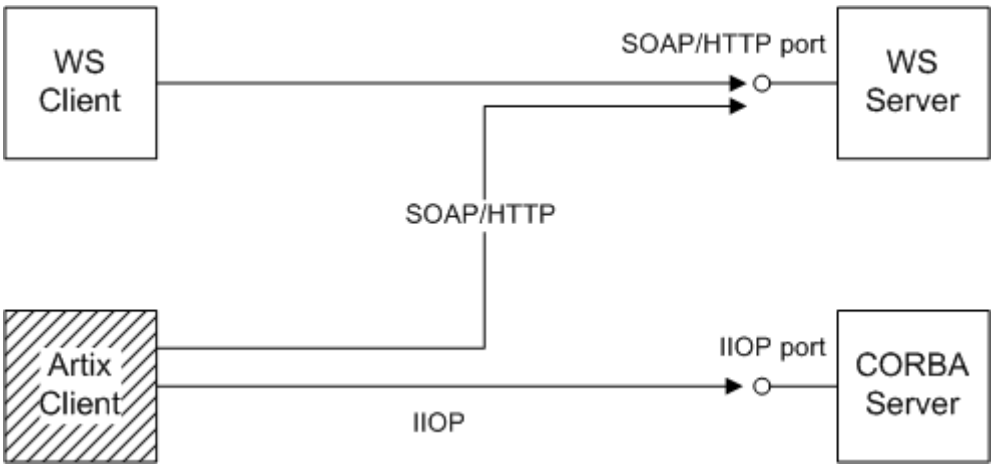

Figure 8: *Replacing the CORBA Client by an Artix Client*

Advantages and disadvantages

This scenario offers the following advantages:

- Compatible with any WS server.
- No changes need be made to the WS server.
- Performance is optimized.
- Artix client offers flexibility for future integration.

And the following disadvantage:

- You must re-implement the CORBA client as an Artix client.

Replacing the WS Server by an Artix Server

If you want to exploit the full power of the Artix product, you might find it worthwhile to replace the WS server by re-implementing it as an Artix server. Because Artix supports multiple protocols, an Artix server can easily support present and future integration requirements.

Figure 9 shows an example of an Artix server that is accessed by a WS client and a CORBA client. The Artix server is configured to accept requests both from CORBA clients and WS clients.

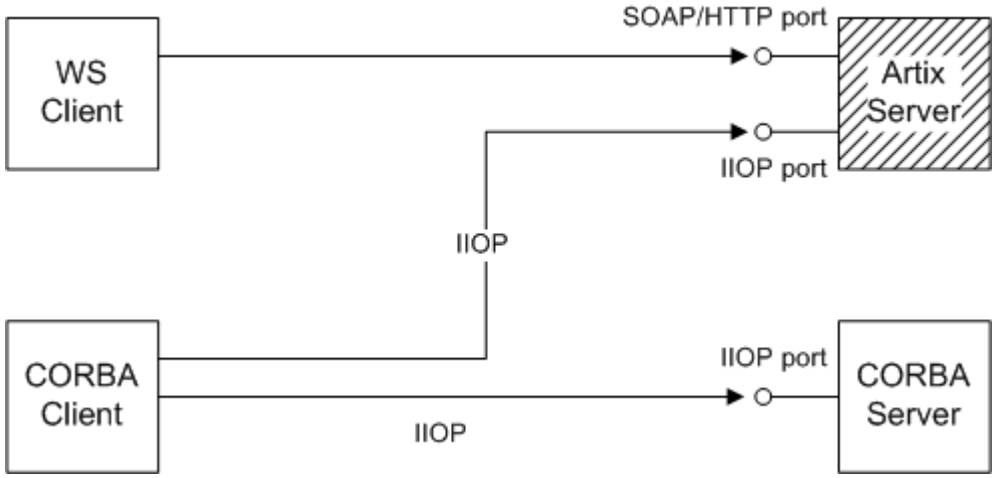

Figure 9: *Replacing the WS Server by an Artix Server*

Advantages and disadvantages

This scenario offers the following advantages:

- Compatible with any CORBA client.
- No changes need be made to the CORBA client.
- Performance is optimized.
- Artix server offers flexibility for future integration.

And the following disadvantage:

- If you have already implemented the WS server using a third-party product, you would have to re-write it as an Artix server.

Exposing a Web Service as a CORBA Service

This chapter describes how to expose a Web service as a CORBA service using Artix. If the Web Service is implemented using Artix, it is relatively easy to integrate with CORBA; if implemented using a third-party product, integration is made possible using Artix routers.

Converting WSDL to IDL

To convert a WSDL contract to an equivalent OMG IDL interface (or interfaces), perform the following steps:

1. Add CORBA bindings to WSDL.
2. Add CORBA endpoints to WSDL.
3. Generate the IDL.

Location of mapping utility

The `wsdltocorba` utility is located in the following `bin` directory:

- *ArtixInstallDir*`/bin`.

Add CORBA bindings to WSDL

Generate a CORBA binding for each port type that you want to expose as an IDL interface:

- If you want to expose a *single* WSDL port type from the WSDL file, `<`*WSDLFile*`>.wsdl`, enter the following command:

  C++ runtime

  ```
  wsdltocorba -corba -i <PortTypeName> <WSDLFile>.wsdl
  ```

  Where `<`*PortTypeName*`>` refers to the `name` attribute of an existing `portType` element. This command generates a new WSDL file, `<`*WSDLFile*`>-corba.wsdl`.

- If you want to expose *multiple* WSDL port types, you must run the `wsdltocorba` command iteratively, once for each port type. For example:

  C++ runtime

  ```
  wsdltocorba -corba -i <PortType_A> -o <WSDLFile>01.wsdl
     <WSDLFile>.wsdl
  wsdltocorba -corba -i <PortType_B> -o <WSDLFile>02.wsdl
     <WSDLFile>01.wsdl
  wsdltocorba -corba -i <PortType_C> -o <WSDLFile>03.wsdl
     <WSDLFile>02.wsdl
  ...
  ```

Where the `-o` flag (C++ runtime) is used to specify the name of the output file at each stage. Rename the last file in the sequence to *<WSDLFile>*`-corba.wsdl`.

Add CORBA endpoints to WSDL

It is not strictly necessary to add CORBA endpoints to the WSDL at this stage (that is, prior to generating the IDL), but it is convenient to make these modifications to the WSDL contract now.

To add the CORBA endpoints, open the *<WSDLFile>*`-corba.wsdl` file generated in the previous step and add a `service` element for each of the port types you want to expose. For example, a simple CORBA endpoint that is associated with the *<CORBABinding>* binding could have the following form:

```
<definitions name="" targetNamespace="..."
    ...
    ...>
    ...
    <service name="<CORBAServiceName>">
       <port binding="tns:<CORBABinding>"
   name="<CORBAPortName>">
          <corba:address location="file:///greeter.ior"/>
       </port>
    </service>
</definitions>
```

The value of the `location` attribute in the `corba:address` element can be specified as one of the following URL types:

- *File URL*—to configure the Artix server to write an IOR to a file as it starts up, specify the `location` attribute as follows:

  `location="file:///`*<DirPath>*`/`*<IORFile>*`.ior"`

  On Windows platforms, the URL format can indicate a particular drive—for example the `C:` drive—as follows:

  `location="file:///C:/`*<DirPath>*`/`*<IORFile>*`.ior"`

  Note: It is usually simplest to specify the file name using an absolute path. If you specify the file name using a relative path, the location is taken to be relative to the directory the Artix process is started in, *not* relative to the containing WSDL file.

- *corbaname URL*—to configure the Artix server to bind an object reference in the CORBA naming service, specify the `location` attribute as follows:

  `location="corbaname:rir:/NameService#`*StringName*`"`

  Where *StringName* is a name in the CORBA naming service. For more details, see "How an Artix Client Resolves a Name" on page 51.

- *Placeholder IOR*—is appropriate for IORs created dynamically at runtime (for example, IORs created by factory objects). In this case, you should use the special placeholder value, `IOR:`, for the location attribute, as follows:

`location="IOR:"`

Artix then uses the enclosing `service` element as a template for transient object references.

Note: It is also possible to add a CORBA endpoint to the WSDL contract using the `wsdltoservice` command line tool. For details of this command, see the *Command Line Reference* document.

Generate the IDL

Generate an IDL interface for each port type, as follows:

- To generate IDL for a *single* port type, select the relevant CORBA binding, `<CORBABinding>`, from the WSDL and enter the following command:

  C++ runtime

  ```
  wsdltocorba -idl -b <CORBABinding>
     <WSDLFile>-corba.wsdl
  ```

  The output from this command is written to an IDL file, `<WSDLFile>-corba.idl`. If you want to change the name of the IDL output file, you can use the `-o <IDLFileName>` option.
- To generate IDL for *multiple* port types, you must run the mapping utility once for each port type. After generating all of the IDL interfaces individually, you would typically concatenate the output files into a single IDL file.

Exposing an Artix Web Service as a CORBA Service

It is relatively straightforward to expose an Artix Web service as a CORBA service. Essentially, you must add the configuration of the relevant CORBA bindings to the WSDL contract and ensure that the requisite CORBA plug-ins are loaded into the Artix application.

In detail, the steps for exposing an Artix service as a CORBA service are as follows:

1. Convert WSDL to IDL.
2. Write code to activate the CORBA endpoints.
3. Re-build the Artix server.
4. Configure the C++ runtime.

Convert WSDL to IDL

Follow the instructions in "Converting WSDL to IDL" on page 11 to convert your WSDL contract to IDL. The output from this step consists of two files, as follows:

- *Modified WSDL file*—the WSDL contract is modified to include CORBA bindings and CORBA endpoints. The Artix server needs the modified contract to expose the service over CORBA.

- *IDL file*—an IDL file is generated from the modified WSDL. CORBA clients use this IDL file to access the CORBA service exposed by the Artix server.

Write code to activate the CORBA endpoints

In the main function of your application source code, add some code to activate the CORBA endpoints. For example, given the following `service` element in the WSDL contract:

```
<definitions name="" targetNamespace="TargetNameSpace"
    ...
    ...>
    ...
    <service name="CORBAServiceName">
        <port binding="tns:CORBABinding"
   name="CORBAPortName">
            <corba:address location="..."/>
        </port>
    </service>
</definitions>
```

You can activate the ports in the *CORBAServiceName* service by registering a servant object with the Artix Bus.

C++ activation code

In C++, you can activate the service, {*TargetNameSpace*}*CORBAServiceName*, as follows:

```
// C++
IT_Bus::QName service_qname("", "CORBAServiceName",
   "TargetNameSpace")

IT_WSDL::WSDLService* wsdl_service =
    m_bus->get_service_contract(service_name);

if (wsdl_service != 0)
{
    m_bus->register_servant(
        *m_servant,     // Service implementation
        *wsdl_service   // WSDL service node
    );
}
```

Where `m_servant` is an object that implements the WSDL service and `service_qname` is the QName of the WSDL service.

Note: For more details about activating service endpoints and registering servants, see the “Artix Programming Considerations” chapter from *C++ Programmer’s Guide*.

Re-build the Artix server

Before re-building the Artix server executable, you must regenerate the Artix stub files from the modified WSDL contract. In particular, you must ensure that stub code is generated for each of the newly-defined CORBA bindings.

After regenerating the stub files, you can re-build the Artix server.

Configure the C++ runtime

The Artix server must be configured to load the requisite CORBA plug-ins. Example 1 shows how to modify the Artix configuration scope, `artix_srvr_with_corba_binding`, to enable the CORBA bindings.

Example 1: *Artix Configuration Required for a CORBA Binding*

```
# Artix Configuration File

artix_srvr_with_corba_binding {
    ...

    # Modified configuration required for a CORBA binding:
    #
1   orb_plugins = [..., "iiop_profile", "giop", "iiop""];
2   binding:client_binding_list =
   ["OTS+POA_Coloc","POA_Coloc","OTS+GIOP+IIOP","GIOP+IIOP"];

3   plugins:iiop_profile:shlib_name = "it_iiop_profile";
    plugins:giop:shlib_name = "it_giop";
    plugins:iiop:shlib_name = "it_iiop";
};
```

The preceding Artix configuration can be explained as follows:

1. Edit the ORB plug-ins list, adding the plug-ins needed to support CORBA bindings. The following additional plug-ins are needed:
   - `iiop_profile`, `giop`, and `iiop` plug-ins—provide support for the Internet Inter-ORB Protocol (IIOP), which is used by CORBA.
2. You should ensure that the `binding:client_binding_list` (either within this scope or in the nearest enclosing scope) includes bindings with the `GIOP+IIOP` protocol combination. The client binding list shown here is a typical default setting.
3. For each of the additional plug-ins you must specify the *root name* of the shared library (or DLL on Windows) that contains the plug-in code. The requisite `plugins:<plugin_name>:shlib_name` entries can be copied from the root scope of the Artix configuration file, `artix.cfg`.

   You can optionally specify additional configuration settings for the plug-ins at this point (see the ***Artix Configuration Reference*** for more details).

Exposing a Non-Artix Web Service as a CORBA Service

If you want to expose a non-Artix Web service as a CORBA service, you must deploy a standalone Artix router that acts as a bridge between CORBA clients and the Web services server.

Standalone CORBA-to-SOAP Router Scenario

Figure 10 shows an overview of a standalone CORBA-to-SOAP router. In this scenario, the router is packaged as a standalone application, which acts as a bridge between the CORBA client and the Web services server. The standalone router is responsible for converting incoming CORBA requests into outgoing requests on the Web services server. Replies from the Web services server are converted into CORBA replies by the router and sent back to the client.

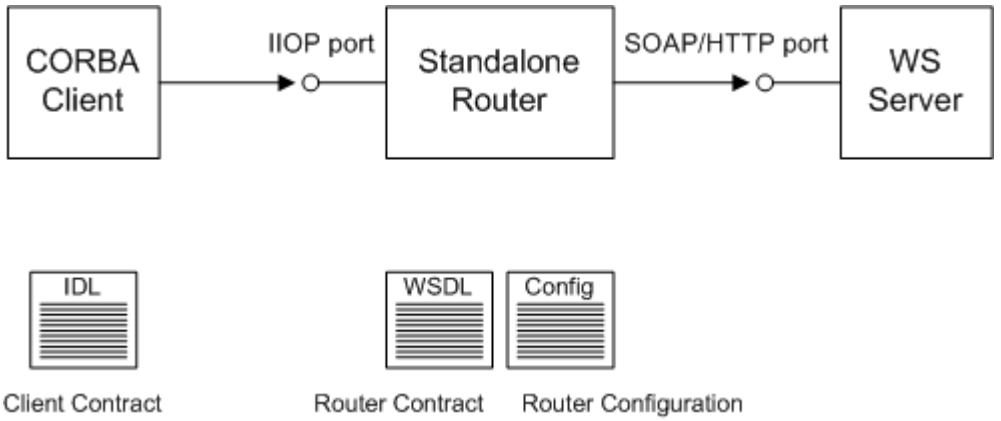

Figure 10: *Standalone Artix Router*

Container

The Artix container, `it_container`, is an executable that can be used to run any of the standard Artix services. The functionality of the container is determined by the plug-ins it loads at runtime.

In this scenario, the container is configured to load the `router` plug-in (along with some other plug-ins) so that it functions as a standalone router.

Modifications to CORBA server

When using a standalone Artix router, no modifications need be made to the CORBA server.

Elements required for this scenario

The following elements are required to implement this scenario:

- IDL interface for clients.
- WSDL contract for the standalone router.
- Artix configuration file for the standalone router.

Configuring and Running a Standalone CORBA-to-SOAP Router

This section describes how to configure and run a standalone router that acts as a bridge between CORBA clients and a SOAP/HTTP Web services server. The following steps are described:

1. Convert WSDL to IDL.
2. Generate the router.wsdl file.
3. Create the Artix configuration.
4. Run the standalone router.

Convert WSDL to IDL

Follow the instructions in "Converting WSDL to IDL" on page 11 to convert your WSDL contract to IDL and to generate CORBA bindings and CORBA endpoints in the WSDL contract. The output from this step is a modified WSDL file, `<`*WSDLFile*`>.wsdl`, and an IDL file.

Generate the router.wsdl file

To generate the `router.wsdl` file, you need to augment the `<`*WSDLFile*`>.wsdl` file from the previous step. Specifically, you must add the requisite bindings and endpoints for the second leg of the route, which goes from the router to the SOAP Web service.

1. *Generate CORBA bindings and CORBA endpoints*—if you followed the steps in "Converting WSDL to IDL" on page 11, the `<`*WSDLFile*`>.wsdl` file already contains the relevant CORBA bindings and CORBA endpoints.
2. *Generate SOAP bindings*—generate a SOAP binding for each port type that is exposed as an IDL interface. The router acts like a SOAP client with respect to the SOAP Web services server.

   If the router needs to access a *single* WSDL port type, generate a SOAP binding with the following command:

   ```
   > wsdltosoap -i <PortTypeName> -b <BindingName> <WSDLFile>.wsdl
   ```

Where <*PortTypeName*> refers to the `name` attribute of an existing `portType` element and <*BindingName*> is the name to be given to the newly generated SOAP binding. This command generates a new WSDL file, <*WSDLFile*>`-soap.wsdl`.

If the router needs to access *multiple* WSDL port types, you must run the `wsdltosoap` command iteratively, once for each port type. For example:

```
> wsdltosoap -i <PortType_A> -b <Binding_A> -o <WSDLFile>01.wsdl
   <WSDLFile>.wsdl
> wsdltosoap -i <PortType_B> -b <Binding_B> -o <WSDLFile>02.wsdl
   <WSDLFile>01.wsdl
> wsdltosoap -i <PortType_C> -b <Binding_C> -o <WSDLFile>03.wsdl
   <WSDLFile>02.wsdl
...
```

Where the `-o` <*FileName*> flag specifies the name of the output file. At the end of this step, rename the WSDL file to `router.wsdl`.

3. Add SOAP endpoints—add a `service` element for each of the port types you want to expose. For example, a simple SOAP endpoint could have the following form:

```
<definitions name="" targetNamespace="..."
    ...
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:http-conf="http://schemas.iona.com/transports/http/
    configuration"
    ...>
    ...
    <service name="<SOAPServiceName>">
        <port binding="tns:<SOAPBinding>"
   name="<SOAPPortName>">
            <soap:address location="http://localhost:9000"/>
            <http-conf:client/>
            <http-conf:server/>
        </port>
    </service>
</definitions>
```

In the preceding example, you must add a line that defines the `http-conf` namespace prefix in the `<definitions>` tag.

The most important setting in the SOAP port is the `location` attribute of the `soap:address` element, which can be set to one of the following HTTP URLs:

- Explicit HTTP URL—if a particular service is provided at a fixed address, you can specify the <*hostname*> and <*port*> values explicitly.

  `location="http://`<*hostname*>`:`<*port*>

- Placeholder HTTP URL—if a service is created dynamically at runtime, you should specify a transient HTTP URL, as follows:

  `location="http://localhost:0`

  At runtime, the placeholder URL is replaced by an explicit address. Artix then treats the enclosing service element

as a template, allowing multiple transient services to be created at runtime.

Note: It is also possible to add a SOAP endpoint to the WSDL contract using the `wsdltoservice` command line tool. For details of this command, see the ***Command Line Reference*** document.

4. Add a route for each exposed port type—for each port type, you need to set up a route to translate incoming CORBA requests into outgoing SOAP requests. For example, the following route definition instructs the router to map incoming CORBA request messages to a SOAP/HTTP endpoint.

```
<definitions name="" targetNamespace="TargetNamespaceURI"
    ...
    xmlns:tns="TargetNamespaceURI"
    xmlns:ns1="http://schemas.iona.com/routing"
    ...>
    ...
    <ns1:route name="route_0">
        <ns1:source      service="tns:<CORBAServiceName>"
                         port="<CORBAPortName>"/>
        <ns1:destination service="tns:<SOAPServiceName>"
                         port="<SOAPPortName>"/>
    </ns1:route>
</definitions>
```

In the preceding example, you must add a line that defines the `ns1` namespace prefix in the `<definitions>` tag.

The `ns1:source` element identifies the CORBA endpoint in the router that receives incoming requests from a client. The `ns1:destination` element identifies the SOAP/HTTP endpoint in the Orbix server to which outgoing requests are routed.

Note: Generally, when defining routes, if the `location` of the source endpoint is a placeholder, the `location` of the destination endpoint should *also* be a placeholder.

5. Check that you have added all the namespaces that you need—for a typical CORBA to SOAP/HTTP route, you typically need to add the following namespaces (in addition to the namespaces already generated by default):

```
<definitions name="" targetNamespace="TargetNamespaceURI"
    ...
    xmlns:tns="TargetNamespaceURI"
    xmlns:ns1="http://schemas.iona.com/routing"
    xmlns:http-conf="http://schemas.iona.com/transports/http/
    configuration"
    xmlns:wsa="http://www.w3.org/2005/08/addressing"
    ...>
    ...
</definitions>
```

Create the Artix configuration

Example 2 shows a suitable configuration for a standalone router that maps incoming CORBA requests to outgoing SOAP/HTTP requests.

Example 2: *Artix Configuration Suitable for a Standalone Artix Router*

```
# Artix Configuration File

# Global configuration scope
...

standalone_router {
    # Configuration for standalone router:
    #
    orb_plugins = ["xmlfile_log_stream", "iiop_profile",
   "giop", "iiop", "soap", "at_http", "routing"];

    plugins:routing:wsdl_url="../../etc/router.wsdl";

    plugins:soap:shlib_name = "it_soap";
    plugins:http:shlib_name = "it_http";
    plugins:at_http:shlib_name = "it_at_http";
    plugins:routing:shlib_name = "it_routing";

    # Uncommment these lines for interoperability with Orbix 3.3

   #policies:giop:interop_policy:negotiate_transmission_codeset
   = "false";
    #policies:giop:interop_policy:send_principal = "true";
    #policies:giop:interop_policy:send_locate_request =
   "false";
};
```

The preceding Artix configuration can be explained as follows:

1. The basic configuration settings needed by the Artix container process are inherited from the global configuration scope.
2. Edit the ORB plug-ins, adding the requisite Artix plug-ins to the list. In this example, the following plug-ins are needed:
   - `xmlfile_log_stream` plug-in—enables logging to an XML file.
   - `iiop_profile`, `giop`, and `iiop` plug-ins—enables the IIOP protocol (used by CORBA).
   - `soap` plug-in—enables the router to send and receive SOAP messages.
   - `at_http` plug-in—enables the router to send and receive messages over the HTTP transport.
   - `routing` plug-in—contains the core of the Artix router.

   If you plan to use other bindings and transports, you might need to add some other Artix plug-ins instead.
3. The `plugins:routing:wsdl_url` setting specifies the location of the router WSDL contract (see "Converting WSDL to IDL" on page 11). The URL can be a relative filename (as here) or a general `file:` URL.

4. To load the Artix plug-ins, you must specify the *root name* of the shared library (or DLL on Windows) that contains the plug-in code. The requisite `plugins:<plugin_name>:shlib_name` entries can be copied from the root scope of the Artix configuration file, `artix.cfg`.

   You can also specify additional plug-in configuration settings at this point (see the *Artix Configuration Reference* for more details).
5. If the router needs to integrate with Orbix 3.3 CORBA clients, you should uncomment these lines to enable interoperability. For more details about these configuration settings, see the *Artix Configuration Reference*.

> **Note:** These interoperability settings might also be useful for integrating with other third-party ORB products. See the ***Artix Configuration Reference*** for more details.

Run the standalone router

Run the standalone router by invoking the container, `it_container`, passing the router's BUS name as a command-line parameter (the BUS name is identical to the name of the router's configuration scope).

For example, to run the router configured in Example 2 on page 20, enter the following at a command prompt:

```
it_container -BUSname standalone_router
```

Using an Orbix 3.3 Client to Access an Artix Server

This section gives a summary of the problems that might occur when you try to compile an Artix-generated IDL file (generated by the `wsdltocorba` tool) using the Orbix 3.3 IDL compiler.

Because the Orbix 3.3 product was designed to conform to the CORBA 2.1 specification (which is an earlier version of the CORBA specification than that used for Artix) there are some differences between the conventions used in Orbix 3.3 IDL files and the conventions used in Artix IDL files.

> **Note:** The following list of issues is not necessarily exhaustive. This section summarizes only those interoperability issues known about at the time of writing.

Incompatible #pragma macros

The following `#pragma` macros which appear in some standard Artix IDL files are incompatible with Orbix 3.3 and will cause the Orbix 3.3 IDL compiler to report an error:

```
#pragma IT_SystemSpecification
#pragma IT_BeginCBESpecific
```

Data type compatibility

Most of the IDL data types generated by the Artix `wsdltocorba` tool are compatible with Orbix 3.3. But there are some exceptions. The following WSDL data types require workarounds in order to interoperate with the Orbix 3.3 product:

- xsd:dateType type mapping to the TimeBase::UtcT IDL type.
- Complex type derived from a simple type.
- Recursive types.

xsd:dateType type mapping to the TimeBase::UtcT IDL type

Artix uses the `TimeBase::UtcT` type to represent the `xsd:dateTime` XML schema type. To support the `TimeBase::UtcT` type, Artix-generated IDL files contain the following `#include` statement:

```
#include <omg/TimeBase.idl>
```

A problem arises, however, when the Orbix 3.3 IDL compiler attempts to compile the `TimeBase.idl` file, because the `TimeBase.idl` file includes `#pragma` macros that are incompatible with the Orbix 3.3 IDL compiler. To fix this problem, perform the following steps:

1. Make a copy of the `TimeBase.idl` file (the original of this file can be found in the *ArtixInstallDir*`/idl/omg` directory).
2. Edit the copied file to delete the following #pragma macros:

```
#pragma IT_SystemSpecification

#pragma IT_BeginCBESpecific AllJava                    "@@\
@module TimeBase=org.omg"
```

3. Edit the `#include` statement in the main IDL file, to point at the modified copy of the `TimeBase.idl` file.

Complex type derived from a simple type

A problem arises with XML schema complex types that are defined by derivation from a simple type. For example, consider the following schema type, `Document`, that adds a string attribute to a simple string type:

```
<xsd:complexType name="Document">
    <xsd:simpleContent>
        <xsd:extension base="xsd:string">
            <xsd:attribute name="ID" type="xsd:string"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
```

When the `wsdltocorba` utility maps this schema type to IDL, it generates the following struct:

```
// IDL
struct Document {
    string_nil ID;
    string _simpleTypeValue;
};
```

When this IDL sample is passed to the Orbix 3.3 compiler, it fails to compile because the Orbix 3.3 compiler does not allow identifiers that begin with the _ (underscore) character.

To work around this problem, you can manually edit the CORBA binding in the WSDL file, replacing `_simpleTypeValue` by `simpleTypeValue` (removing the underscore character). For example, for the `Document` data type, the CORBA binding defines the following mapping by default:

```
<corba:struct name="Document"
   repositoryID="IDL:Document:1.0"
              type="s:Document">
    <corba:member idltype="ns1:string_nil" name="ID"/>
    <corba:member idltype="corba:string"
                  name="_simpleTypeValue"/>
</corba:struct>
```

To modify the mapping in this case, simply replace `_simpleTypeValue` by `simpleTypeValue` in the preceding code fragment.

Recursive types

The IDL mapping for recursive XML schema types requires the use of forward declared structs in IDL. The forward declared struct is a relatively recent addition to IDL syntax and is not supported by Orbix 3.3. Hence, recursive types are incompatible with Orbix 3.3 clients.

For more details about XML schema recursive types, see "Recursive Types" on page 113.

Accessing an Artix Server Using WSDL Query

Usually, a CORBA client that wants to access a CORBA service would locate the service through a CORBA mechanism—for example, the CORBA naming service (see "Integrating the CORBA Naming Service with Artix" on page 51).

In an enterprise application, however, it is likely that the CORBA service exposed by an Artix server could also be accessed by another Artix program. Alternatively, there might be some technical reasons for preferring to connect two Artix programs using the CORBA protocol. In either of these scenarios, it is possible to locate the CORBA service using an Artix-specific mechanism, the *WSDL publish service*. This enables the client to bypass the CORBA naming service, but it does require that the client knows the host and port of the WSDL publish service.

Configure WSDL publish on the server side

The WSDL publish service is deployed as a plug-in, `wsdl_publish`, on the Artix server side. Example 3 shows the basic configuration of the WSDL publish service for a server that takes its configuration from the `wsdl_publish.server` scope.

Example 3: *Configuration of WSDL Publish in an Artix Server*

```
# Artix Configuration File
wsdl_publish {
    server {
        orb_plugins = ["xmlfile_log_stream",
   "wsdl_publish"];
        plugins:wsdl_publish:publish_port = "4444";
        ...
    };
};
```

Where the WSDL publish plug-in is configured to listen on IP port 4444 for WSDL queries and other requests. For complete details on how to configure the WSDL publish service, see *Configuring and Deploying Artix Solutions*.

WSDL query URL for CORBA services

An Artix client can now retrieve the WSDL contract for a specific WSDL service by downloading the response to the following URL:

`http://`*Host*`:`*Port*`/get_wsdl?service=`*ServiceName*`&scope=`*TargetNameSpace*

Where *Host*`:`*Port* is the IP address of the WSDL publish service, *ServiceName* is the local name of the required service, and *TargetNameSpace* is the namespace in which the service is defined. Although other query URL formats are described in the *Configuring and Deploying Artix Solutions* guide, the preceding format is the only one that works for the CORBA binding.

Note: In addition to the insecure HTTP URL format described here, it is also possible to configure WSDL publish to support the secure HTTPS protocol. For more details, please consult the *Artix Security Guide*.

Artix client configuration

As an example of how to use a WSDL query URL, consider an Artix client that consumes a CORBA service exposed by an Artix server.

Assuming that the client has been programmed to use the usual contract locating mechanism (for example, in C++ the Artix client would be programmed to locate the contract using the

IT_Bus::Bus::get_service_contract() function), you could configure the client to query the WSDL contract by specifying the following configuration:

```
# Artix Configuration File
wsdl_query {
    ...
    client {
        bus:qname_alias:greeter =

   "{http://www.iona.com/hello_world_soap_http}SOAPServic
   e";
        bus:initial_contract:url:greeter =
"http://foo:1001/get_wsdl?service=SOAPService&scope=http:
   //
 www.iona.com/hello_world_soap_http";
    };
};
```

Where the configuration variable bus:qname_alias:*QNameAlias* defines the QName alias, greeter, which acts as a shorthand for the specified QName. The bus:initial_contract:url:greeter configuration variable specifies the query URL that the client uses to retrieve the WSDL contract.

Exposing a CORBA Service as a Web Service

This chapter describes how to expose a CORBA service as a Web service using Artix. Different approaches can be taken, depending on whether the back-end CORBA service is implemented using the Orbix 6 product, the Orbix 3.3 product or some other third-party ORB product.

Converting IDL to WSDL

The first step in exposing a CORBA server as a Web service is to convert the CORBA server's IDL into a WSDL contract. For all of the examples presented in this chapter, the following assumptions are made:

- The server's IDL does not feature callbacks.
- Web service clients use the SOAP/HTTP protocol.

WSDL contract files

This subsection describes how to generate the following two WSDL files:

- `router.wsdl` file—deployed along with the embedded router and the Orbix server, the `router.wsdl` file contains all of the router information required to map incoming SOAP requests to outgoing CORBA requests.
- `client.wsdl` file—contains all of the information required by Web services clients to make SOAP/HTTP invocations on the router.

Contents of the router contract

Given that the router has to be capable of routing incoming SOAP requests to outgoing CORBA requests, the router generally must contain the following elements:

- Port types.
- CORBA bindings.
- SOAP bindings.
- CORBA endpoints.
- SOAP/HTTP endpoints.
- Routes from SOAP/HTTP endpoints to CORBA endpoints.

Generate the router contract

To generate a router contract from a given IDL file, <*IDLFile*>.idl, perform the following steps:

1. Generate WSDL from the IDL file—at a command-line prompt, enter:

   ```
   > idltowsdl <IDLFile>.idl
   ```

   This command generates a WSDL file, <*IDLFile*>.wsdl, which contains the following:

   - XSD schema types, generated from the IDL data types.
   - portType elements—a port type for each IDL interface in the source.
   - binding elements—a CORBA binding for each port type.
   - service elements—a CORBA endpoint for each port type

   You might need to specify additional flags to the idltowsdl command utility. Some of the more commonly required options are:

   -r <*ref_schema*> specifies the location of the endpoint references schema. The schema file, wsaddressing.xsd, is located in the *ArtixInstallDir*/schemas directory and on the Internet. The references schema is needed whenever you generate WSDL from IDL that uses object references.

   -a <*corba_address*> specifies a default value for the location attribute in the corba:address elements.

   -unwrap generates doc/literal unwrapped style of WSDL.

   -usetypes generates rpc/literal style of WSDL.

   -3 specifies Orbix 3.3 compatibility mode. Use this option if the IDL file you are converting stems from a legacy Orbix 3.3 application. See "Orbix 3 legacy compatibility" on page 147 for more details.

   The default style of WSDL generated by the idltowsdl utility is doc/literal wrapped.

2. Edit the corba:address elements for each CORBA endpoint—for each CORBA endpoint, you have to specify the location of a CORBA object reference.

   Using your favorite text editor, open the <*IDLFile*>.wsdl file generated in the previous step. Replace the dummy setting, location="...", in each of the corba:address elements, by one of the following location URL settings:

   - *File URL*—if the Orbix server writes an IOR to a file as it starts up, you specify the location attribute as follows:

     ```
     location="file:///<DirPath>/<IORFile>.ior"
     ```

     On Windows platforms, the URL format can indicate a particular drive—for example the C: drive—as follows:

location="file:///C:/<*DirPath*>/<*IORFile*>.ior"

Note: It is usually simplest to specify the file name using an absolute path. If you specify the file name using a relative path, the location is taken to be relative to the directory the Artix process is started in, *not* relative to the containing WSDL file.

- *corbaname URL*—allows you to retrieve an object reference from the CORBA naming service. This setting has the following format:

  location="corbaname:rir:/NameService#*StringName*"

  Where *StringName* is a name in the CORBA naming service. For more details, see "How an Artix Client Resolves a Name" on page 51.
- *Stringified IOR*—if you know that the Orbix server's IOR is not going to change for some time, you can paste the stringified IOR directly into the location attribute, as follows:

  location="IOR:000000..."
- *Placeholder IOR*—is appropriate for IORs created dynamically at runtime (for example, IORs created by factory objects). In this case, you should use the special placeholder value, IOR:, for the location attribute, as follows:

  location="IOR:"

  Artix uses the enclosing service element as a template for transient object references.

For example, if your Orbix server writes an IOR to the file, /tmp/app_iors/hello_world_service.ior, you can use it to specify the endpoint location as follows:

```
<service name="HelloWorldCORBAService">
    <port binding="tns:HelloWorldCORBABinding" name="HelloWorldCORBAPort">
      <corba:address location="file:///tmp/app_iors/hello_world_service.ior"/>
    </port>
</service>
```

3. Generate SOAP bindings—generate a SOAP binding for each port type that you want to expose as a Web service. If you want to expose a single WSDL port type, enter the following command:

   > wsdltosoap -i <*PortTypeName*> -b <*BindingName*> <*IDLFile*>.wsdl

   Where <*PortTypeName*> refers to the name attribute of an existing portType element and <*BindingName*> is the name to be given to the newly generated SOAP binding. This command generates a new WSDL file, <*IDLFile*>-soap.wsdl.

   If you want to expose *multiple* WSDL port types, you must run the wsdltosoap command iteratively, once for each port type. For example:

   > wsdltosoap -i <*PortType_A*> -b <*Binding_A*> -o <*IDLFile*>01.wsdl <*IDLFile*>.wsdl
   > wsdltosoap -i <*PortType_B*> -b <*Binding_B*> -o <*IDLFile*>02.wsdl <*IDLFile*>01.wsdl

```
> wsdltosoap -i <PortType_C> -b <Binding_C> -o <IDLFile>03.wsdl
   <IDLFile>02.wsdl
...
```

Where the `-o <FileName>` flag specifies the name of the output file. At the end of this step, rename the WSDL file to `router.wsdl`.

4. Add SOAP endpoints—add a `service` element for each of the port types you want to expose. For example, a simple SOAP endpoint could have the following form:

```
<definitions name="" targetNamespace="..."
    ...
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:http-conf="http://schemas.iona.com/transports/http/
    configuration"
    ...>
    ...
    <service name="<SOAPServiceName>">
        <port binding="tns:<SOAPBinding>"
   name="<SOAPPortName>">
            <soap:address location="http://localhost:9000"/>
            <http-conf:client/>
            <http-conf:server/>
        </port>
    </service>
</definitions>
```

In the preceding example, you must add a line that defines the `http-conf` namespace prefix in the `<definitions>` tag.

The most important setting in the SOAP port is the `location` attribute of the `soap:address` element, which can be set to one of the following HTTP URLs:

- Explicit HTTP URL—if a particular service is meant to listen on a fixed address, you can specify the `<hostname>` and `<port>` values explicitly.

  `location="http://<hostname>:<port>`

- Placeholder HTTP URL—if a service is meant to be created dynamically at runtime, you should specify a transient HTTP URL, as follows:

  `location="http://localhost:0`

  At runtime, the placeholder URL is replaced by an explicit address when the service is created. Artix treats the enclosing `service` element as a template, allowing multiple transient services to be created at runtime.

Note: It is also possible to add a SOAP endpoint to the WSDL contract using the `wsdltoservice` command line tool. For details of this command, see the ***Command Line Reference*** document.

5. Add a route for each exposed port type—for each port type, you need to set up a route to translate incoming SOAP requests into outgoing CORBA requests. For example, the following route definition instructs the router to map incoming SOAP/HTTP request messages to a CORBA endpoint.

```
<definitions name="" targetNamespace="TargetNamespaceURI"
    ...
    xmlns:tns="TargetNamespaceURI"
    xmlns:ns1="http://schemas.iona.com/routing"
    ...>
    ...
    <ns1:route name="route_0">
        <ns1:source      service="tns:<SOAPServiceName>"
                         port="<SOAPPortName>"/>
        <ns1:destination service="tns:<CORBAServiceName>"
                         port="<CORBAPortName>"/>
    </ns1:route>
</definitions>
```

In the preceding example, you must add a line that defines the `ns1` namespace prefix in the `<definitions>` tag.

The `ns1:source` element identifies the SOAP/HTTP endpoint in the router that receives incoming requests from a client. The `ns1:destination` element identifies the CORBA endpoint in the Orbix server to which outgoing requests are routed.

Note: Generally, when defining routes, if the location of the source endpoint is a placeholder, the location of the destination endpoint should *also* be a placeholder.

6. Check that you have added all the namespaces that you need—for a typical SOAP/HTTP to CORBA route, you typically need to add the following namespaces (in addition to the namespaces already generated by default):

```
<definitions name="" targetNamespace="TargetNamespaceURI"
    ...
    xmlns:tns="TargetNamespaceURI"
    xmlns:ns1="http://schemas.iona.com/routing"
    xmlns:http-conf="http://schemas.iona.com/transports/http/
    configuration"
    xmlns:wsa="http://www.w3.org/2005/08/addressing"
    ...>
    ...
</definitions>
```

7. Include the WS-Addressing schema (if required)—if your IDL passes any object references (for example, as parameters or return values), the corresponding WSDL contract needs to include the WS-Addressing schema to represent the object references.

For example, assuming that the `wsaddressing.xsd` schema file is stored in the same directory as `router.wsdl`, you can include the WS-Addressing schema in the router contract as follows:

```
<definitions name="" targetNamespace="TargetNamespaceURI"
    ...>
    <types>
        <schema targetNamespace="..."  ...>
           <import namespace="http://www.w3.org/2005/08/addressing"
               schemaLocation="wsaddressing.xsd"/>
            ...
        </schema>
    </types>
    ...
</definitions>
```

The original copy of the `wsaddressing.xsd` schema file is located in the *ArtixInstallDir*`/schemas` directory.

router.wsdl file contents

For example, if the router contract contains a single port type, the contents of `router.wsdl` would have the following outline:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="" targetNamespace="TargetNamespaceURI"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:corba="http://schemas.iona.com/bindings/corba"
xmlns:corbatm="http://schemas.iona.com/typemap/corba/cdr_over_i
   iop.idl"
    xmlns:wsa="http://www.w3.org/2005/08/addressing"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:http-conf="http://schemas.iona.com/transports/http/config
   uration"
    xmlns:ns1="http://schemas.iona.com/routing"
    xmlns:tns="TargetNamespaceURI"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsd1="http://schemas.iona.com/idltypes/cdr_over_iiop.idl"
   >
    <types>
        ...
    </types>
    <message name="..."/>
    ...

    <portType name="<PortTypeName>">
        ...
    </portType>

    <binding name="<CORBABindingName>"
             type="tns:<PortTypeName>">
        ...
    </binding>

    <binding name="<SOAPBindingName>"
             type="tns:<PortTypeName>">
        ...
    </binding>
```

```
    <service name="<CORBAServiceName>">
        ...
    </service>

    <service name="<SOAPServiceName>">
        ...
    </service>

    <ns1:route name="route_0">
        <ns1:source      service="tns:<SOAPServiceName>"
                         port="<SOAPPortName>"/>
        <ns1:destination service="tns:<CORBAServiceName>"
                         port="<CORBAPortName>"/>
    </ns1:route>
</definitions>
```

Generate the client contract

The client WSDL contract is a modified copy of the router contract containing only those details of the contract that are relevant to the client. To generate the client contract, perform the following steps:

1. Copy the `router.wsdl` file to `client.wsdl`.
2. Edit the `client.wsdl` file to remove redundant elements. That is, you should remove the following:
   - CORBA `binding` elements.
   - CORBA `service` elements.
   - `route` elements.

   You could also optionally remove some of the redundant namespace definitions, such as `corba`, `corbatm`, and `ns1`.

client.wsdl file contents

For example, if the client contract contains a single port type, the contents of `client.wsdl` would have the following outline:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="" targetNamespace="TargetNamespaceURI"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:corba="http://schemas.iona.com/bindings/corba"
xmlns:corbatm="http://schemas.iona.com/typemap/corba/cdr_over_i
   iop.idl"
    xmlns:wsa="http://www.w3.org/2005/08/addressing"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:http-conf="http://schemas.iona.com/transports/http/config
   uration"
    xmlns:ns1="http://schemas.iona.com/routing"
    xmlns:tns="TargetNamespaceURI"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsd1="http://schemas.iona.com/idltypes/cdr_over_iiop.idl"
   >
    <types>
        ...
    </types>
```

```
    <message name="..."/>
    ...

    <portType name="<PortTypeName>">
        ...
    </portType>

    <binding name="<SOAPBindingName>"
             type="tns:<PortTypeName>">
        ...
    </binding>

    <service name="<SOAPServiceName>">
        ...
    </service>
</definitions>
```

Embedding Artix in a CORBA Service

If you want to expose an Orbix 6 CORBA server as a Web service, you have the option of embedding Artix directly in the CORBA server.

This embedding is possible because Artix (C++ runtime) and Orbix are both built using the same framework:the Adaptive Runtime Technology (ART). Using the ART framework, it is possible to run Artix and Orbix in the same process just by loading the appropriate set of plug-ins needed by each product.

Embedded Router Scenario

Figure 11 shows an overview of an Artix router embedded in a CORBA server. In this scenario, the CORBA service is exposed as a Web service that supports SOAP over HTTP. The embedded router is responsible for converting incoming SOAP/HTTP requests into colocated requests on the CORBA server. Any replies from the CORBA server are then converted into SOAP/HTTP replies by the router and sent back to the client.

Note: Embedding an Artix router is an option that is *only* available to Orbix 6 based CORBA applications. In general, the most straightforward way to build these applications is to use the Orbix libraries included with the Artix product. If you need to link with libraries taken directly from an Orbix distribution, you must take care to ensure that these libraries are binary compatible with Artix.

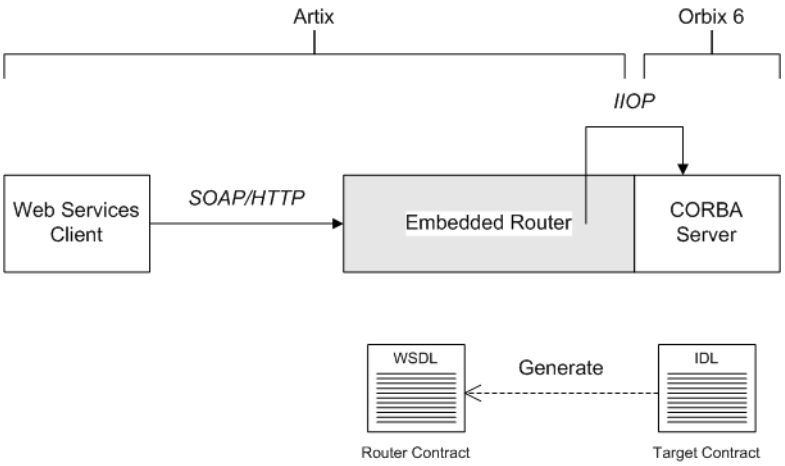

Figure 11: *Artix Router Embedded in a CORBA Server*

Modifications to CORBA server

The following changes must be made to the CORBA server to embed the Artix router:

- Code changes—*No*.
- Re-compilation—*No*.
- Configuration—modify the Orbix configuration file.

Elements required for this scenario

The following elements are required to implement this scenario:

- WSDL contract for clients.
- WSDL contract for the embedded router.
- Modified Orbix configuration file for the CORBA server.

Embedding a Router in the CORBA Server

This section describes how to embed a router in a CORBA server. The embedded router enables the CORBA server to receive requests from a SOAP/HTTP Web services client. The following steps are described:

- Convert IDL to WSDL.
- Deploy the requisite WSDL files.
- Edit the Artix configuration.

Convert IDL to WSDL

Use the Artix utilities to generate two WSDL files, `router.wsdl` and `client.wsdl`, from the CORBA server's IDL interface. For details of how to convert the IDL file to WSDL, see "Converting IDL to WSDL" on page 27.

Deploy the requisite WSDL files

Deploy the following WSDL files on the CORBA server host:

- `router.wsdl`—the router contract, which describes the route for converting SOAP/HTTP requests into CORBA requests.
- `wsaddressing.xsd`—the schema that defines the `wsa:EndpointReferenceType` data type, which Artix uses to represent object references.

  The WS-Addressing schema is usually (but not always) required on the server side. If your IDL does not pass object endpoint references as parameters or return values, however, you do not need to deploy this file.

Edit the Artix configuration

Given that your CORBA server is configured by a particular configuration scope, `orbix_srvr_with_embeded_router`, Example 4 shows how to modify the server configuration to embed an Artix router.

Example 4: *Artix Configuration Suitable for an Embedded Artix Router*

```
# Artix Configuration File

orbix_srvr_with_embedded_router {
    ...

    # Modified configuration required for embedded router:
    #
1   orb_plugins = [..., "soap", "at_http", "routing", "bus_loader"];
2   binding:client_binding_list = ["OTS+GIOP+IIOP",
   "GIOP+IIOP"];

3   plugins:routing:wsdl_url="../../etc/router.wsdl";

4   plugins:soap:shlib_name = "it_soap";
    plugins:http:shlib_name = "it_http";
    plugins:at_http:shlib_name = "it_at_http";
    plugins:routing:shlib_name = "it_routing";
    plugins:bus_loader:shlib_name = "it_bus_loader";

5   share_variables_with_internal_orb = "false";
};
```

The preceding Artix configuration can be explained as follows:

1. Edit the ORB plug-ins, adding the requisite Artix plug-ins to the list. In this example, the following plug-ins are needed:
   - `soap` plug-in—enables the router to send and receive SOAP messages.
   - `at_http` plug-in—enables the router to send and receive messages over the HTTP transport.
   - `routing` plug-in—contains the core of the Artix router.

- `bus_loader` plug-in—triggers the Artix Bus initialization step. This plug-in is needed only when you are loading Artix plug-ins into a non-Artix application.

Note: In Artix 3.0, Artix plug-ins were refactored to cleanly separate the ORB initialization step from the Artix Bus initialization step. Usually, in an Artix application, `IT_Bus::init()` triggers the Bus initialization step. In this example, however, the CORBA server never calls `IT_Bus::init()`. Therefore, the `bus_loader` plug-in is needed to finish the initialization of the Artix plug-ins.

If you plan to use other bindings and transports, you might need to add some other Artix plug-ins instead.

2. The Artix embedded router is *not* compatible with the `POA_Coloc` interceptor. Therefore you must edit the server's `binding:client_binding_list` entry to remove any bindings containing the `POA_Coloc` interceptor.

   For example, if the client binding list is defined as follows:

   ```
   binding:client_binding_list =
      ["OTS+POA_Coloc","POA_Coloc","OTS+GIOP+IIOP","GIOP+IIOP"];
   ```

   You would replace it with the following list:

   ```
   binding:client_binding_list = ["OTS+GIOP+IIOP","GIOP+IIOP"];
   ```

   Note: If the `binding:client_binding_list` variable does not appear explicitly in the server's configuration scope, try to find it in the next enclosing scope (or the scope that is nearest to the server's configuration scope) and copy it into the server's scope.

   If you do not purge the `POA_Coloc` entries from the client binding list, clients that attempt to access the server through the router will receive a `CORBA::UNKNOWN` exception.

3. The `plugins:routing:wsdl_url` setting specifies the location of the router WSDL contract (see "Converting IDL to WSDL" on page 27). The URL can be a relative filename (as here) or a general `file:` URL.

4. In order for Orbix to load the Artix plug-ins, for each plug-in you must specify the *root name* of the shared library (or DLL on Windows) that contains the plug-in code. The requisite `plugins:<`*plugin_name*`>:shlib_name` entries can be copied from the root scope of the Artix configuration file, `artix.cfg`.

   You can also specify additional configuration settings for the Artix plug-ins at this point (see the *Artix Configuration Reference* for more details).

5. In certain circumstances, Orbix creates an internal ORB instance (for example, during initialization). To prevent the settings from the current scope being used by the internal ORBs—specifically, to prevent the internal ORB from loading Artix plug-ins—you should set the `share_variables_with_internal_orb` configuration variable to `false`.

Exposing an Orbix 3.3 or Non-Orbix Service as a Web Service

If you want to expose an Orbix 3.3 or non-Orbix CORBA server as a Web service, it is generally necessary to deploy a standalone Artix router that acts as a bridge between Web services clients and the CORBA server. Using a standalone router is a non-intrusive integration approach that should work with any CORBA back-end.

Standalone SOAP-to-CORBA Router Scenario

Figure 12 shows an overview of a standalone router. In this scenario, the router is packaged as a standalone application, which acts as a bridge between the Web services client and the CORBA server. The standalone router is responsible for converting incoming SOAP/HTTP requests into outgoing requests on the CORBA server. Replies from the CORBA server are converted into SOAP/HTTP replies by the router and sent back to the client.

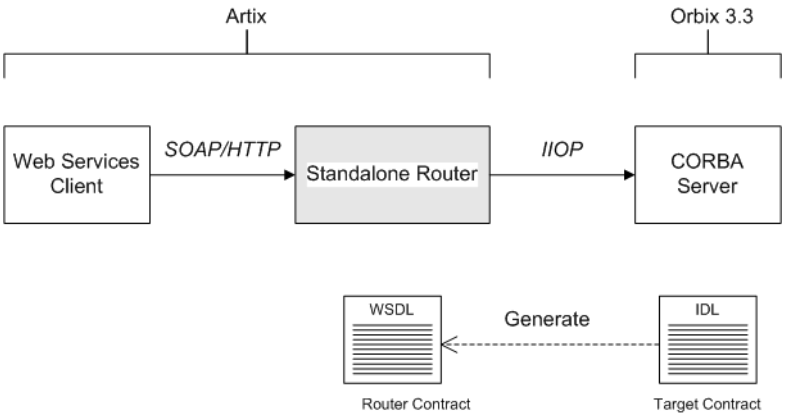

Figure 12: *Standalone Artix Router*

Container

The Artix container, `it_container`, is an application that can be used to run any of the standard Artix services. The functionality of the container is determined by the plug-ins it loads at runtime.

In this scenario, the container is configured to load the `router` plug-in (along with some other plug-ins) so that it functions as a standalone router.

Modifications to CORBA server

When using a standalone Artix router, no modifications need be made to the CORBA server.

Elements required for this scenario

The following elements are required to implement this scenario:

- WSDL contract for clients.
- WSDL contract for the standalone router.
- Artix configuration file for the standalone router.

Configuring and Running a Standalone SOAP-to-CORBA Router

This section describes how to configure and run a standalone router that acts as a bridge between a SOAP/HTTP Web services client and a CORBA server. The following steps are described:

- Convert IDL to WSDL.
- Deploy the requisite WSDL files.
- Create the Artix configuration.
- Run the standalone router.

Convert IDL to WSDL

Use the Artix utilities to generate two WSDL files, `router.wsdl` and `client.wsdl`, from the CORBA server's IDL interface. For details, see "Converting IDL to WSDL" on page 27.

Deploy the requisite WSDL files

Deploy the following WSDL files on the standalone router host:

- `router.wsdl`—the router contract, which describes the route for converting SOAP/HTTP requests into CORBA requests.
- `wsaddressing.xsd`—the schema that defines the `wsa:EndpointReferenceType` data type, which Artix uses to represent object references.

  The WS-Addressing schema is usually (but not always) required on the server side. If your IDL does not pass object references as parameters or return values, however, you do not need to deploy this file.

Create the Artix configuration

Example 5 shows a suitable configuration for a standalone router that maps incoming SOAP/HTTP requests to outgoing CORBA requests.

Example 5: *Artix Configuration Suitable for a Standalone Artix Router*

```
# Artix Configuration File

standalone_router {
    # Configuration for standalone router:
```

Example 5: *Artix Configuration Suitable for a Standalone Artix Router*

```
    #
1   orb_plugins = ["xmlfile_log_stream", "iiop_profile",
   "giop", "iiop", "soap", "at_http", "routing"];

2   plugins:routing:wsdl_url="../../etc/router.wsdl";

3   plugins:soap:shlib_name = "it_soap";
    plugins:http:shlib_name = "it_http";
    plugins:at_http:shlib_name = "it_at_http";
    plugins:routing:shlib_name = "it_routing";

    # Uncomment these lines for interoperability with Orbix 3.3
4
   #policies:giop:interop_policy:negotiate_transmission_codeset
   = "false";
    #policies:giop:interop_policy:send_principal = "true";
    #policies:giop:interop_policy:send_locate_request =
   "false";
};
```

The preceding Artix configuration can be explained as follows:

1. Edit the ORB plug-ins, adding the requisite Artix plug-ins to the list. In this example, the following plug-ins are needed:
   - `xmlfile_log_stream` plug-in—enables logging to an XML file.
   - `iiop_profile`, `giop`, and `iiop` plug-ins—enables the IIOP protocol (used by CORBA).
   - `soap` plug-in—enables the router to send and receive SOAP messages.
   - `at_http` plug-in—enables the router to send and receive messages over the HTTP transport.
   - `routing` plug-in—contains the core of the Artix router.

   If you plan to use other bindings and transports, you might need to add some other Artix plug-ins instead.
2. The `plugins:routing:wsdl_url` setting specifies the location of the router WSDL contract (see "Converting IDL to WSDL" on page 27). The URL can be a relative filename (as here) or a general `file:` URL.
3. To load the Artix plug-ins, you must specify the *root name* of the shared library (or DLL on Windows) that contains the plug-in code. The requisite `plugins:<`*plugin_name*`>:shlib_name` entries can be copied from the root scope of the Artix configuration file, `artix.cfg`.

   You can also specify additional plug-in configuration settings at this point (see the *Artix Configuration Reference* for more details).

4. If the router needs to integrate with an Orbix 3.3 CORBA server, you should uncomment these lines to enable interoperability. For more details about these configuration settings, see the *Artix Configuration Reference*.

> **Note:** These interoperability settings might also be useful for integrating with other third-party ORB products. See the *Artix Configuration Reference* for more details.

Run the standalone router

Run the standalone router by invoking the container, `it_container`, passing the router's ORB name as a command-line parameter (the ORB name is identical to the name of the router's configuration scope).

For example, to run the router configured in Example 5 on page 39, enter the following at a command prompt:

```
it_container -BUSname standalone_router
```

CORBA-to-CORBA Routing

This chapter describes some special routing options that are available when the source endpoint and the destination endpoint in a route are both based on the CORBA binding.

Bypassing the Router

Specifically for the CORBA binding, the Artix router supports an option to redirect incoming client connections so that the clients connect directly to the target server, bypassing the router. This option is only available, if both the client and the target server are CORBA-based. Bypassing the router enables you to achieve optimum efficiency for a CORBA-to-CORBA route, but this option also has some interactions with other router features.

Basic Bypass Scenario

Bypass routing is a CORBA-specific feature that exploits the *location forwarding* feature of the General Inter-ORB Protocol (GIOP). Location forwarding is based on specific GIOP message types, which enable CORBA services to redirect incoming connections to alternative destinations. Figure 13 gives an overview of a basic bypass routing scenario.

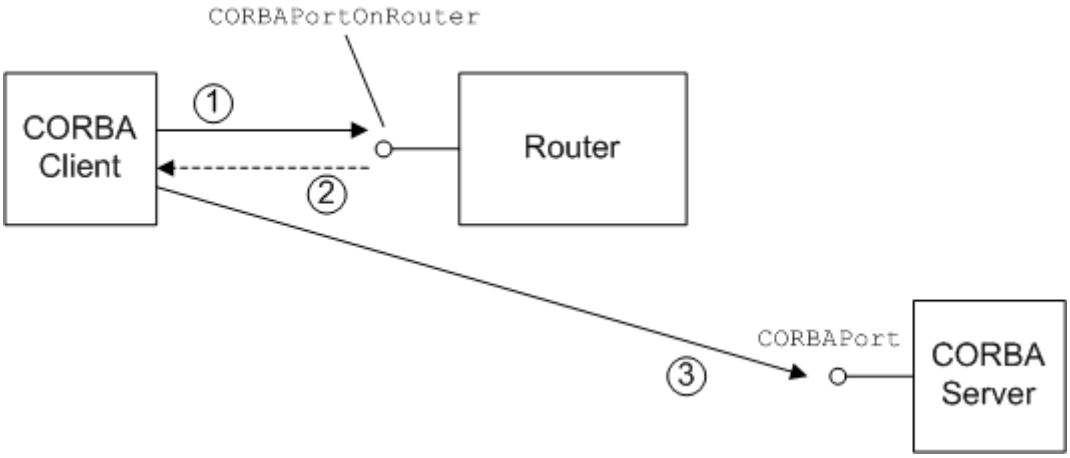

Figure 13: *Basic Bypass Routing Scenario*

Scenario steps

The basic bypass routing scenario shown in Figure 13 can be described as follows:

1. The CORBA client sends a GIOP request message to the `CORBAPortOnRouter` endpoint.
2. The router sends a location forward reply (a special reply type defined by GIOP), which contains the interoperable object reference (IOR) for the destination endpoint on the target server.

   Note: Internally, the router converts the address of the destination endpoint to an IOR using the `CORBA::ORB::string_to_object()` function. This affects the semantics of connection establishment.

   For example, if the destination endpoint is specified as a `corbaname` URL, the router would implicitly resolve the name to an IOR (by contacting a CORBA naming service) before sending the location forward reply.

3. The CORBA client uses the received IOR to open a connection *directly* to the destination endpoint on the target server. The client now sends its request messages directly to the destination endpoint on the target.

   Note: This step might also involve sending an additional location forward message. For example, if the destination endpoint is an Orbix server with a `plain_text_key` plug-in, the server might need to look up the incoming object key in the `plain_text_key` plug-in's registry to obtain the complete IOR. This IOR would then be sent back to the client inside a location forward reply.

Interactions with other features

Bypass routing interacts with various other router features, as follows:

- Effect on pass-through.
- Effect on security.
- Incompatibility with fanout.
- Incompatibility with content-based routing.
- Incompatibility with transport attributes.
- Unsuitability for connection concentrator.

Effect on pass-through

Bypass routing and pass-through routing can be enabled simultaneously. If the route is CORBA only (that is, the binding types for the source and destination endpoints are both CORBA), bypass routing takes priority. For non-CORBA binding types, pass-through routing is used.

Effect on security

When bypass routing is enabled, you must ensure that the CORBA client is appropriately configured for opening a secure connection *directly* to the destination endpoint.

It is important also to understand that the router does not provide any protection for the destination endpoint. The CORBA server on the far side of the router must be independently capable of enforcing the level of security that it requires.

Incompatibility with fanout

Bypass routing is *not* compatible with fanout routes (which can be enabled by setting the `multiRoute` attribute to fanout in the `routing:route` element). *A fanout route* denotes a route where each incoming request message propagates to multiple recipients on the destination side of the route.

If fanout is enabled, the router would ignore the bypass setting and implement fanout instead.

Incompatibility with content-based routing

Bypass routing is *not* compatible with content-based routing (which can be configured using the `routing:query` element in the router contract).

If content-based routing is enabled, the router would ignore the bypass setting and implement content-based routing instead.

Incompatibility with transport attributes

Bypass routing is *not* compatible with routes defined using transport attributes (which can be configured using the `routing:transportAttribute` element in the router contract). Transport attributes enable you to specify a route based on the values set in the transport attributes in the message headers.

If transport attributes based routing is enabled, the router would ignore the bypass setting and implement transport attributes based routing instead.

Unsuitability for connection concentrator

A *connection concentrator* is a deployment pattern, where multiple clients connect to the same source endpoint on a router, but there is only a single connection from the router to the destination endpoint. This pattern enables you to reduce the number of connections made to the destination endpoint.

It does not make sense to use the bypass feature with a connection concentrator, because all of the client connections would end up going directly to the destination endpoint.

Configuring router bypass

To enable router bypass, add the following setting to the router's configuration:

```
# Artix Configuration File
bypass_router
{
    plugins:routing:use_bypass = "true";
    ...
};
```

The default is `false`.

Sample route

Example 6 shows an example of a basic bypass route that listens for connection attempts on the `CORBAPortOnRouter` endpoint and then forwards the connections on to the `CORBAPort` endpoint.

Example 6: *Sample Bypass Route*

```
<definitions name="" targetNamespace="TargetNamespaceURI"
    ...
    xmlns:tns="TargetNamespaceURI"
    xmlns:ns1="http://schemas.iona.com/routing"
    ...>
    ...
    <ns1:route name="pass_through_route">
        <ns1:source
   service="tns:CORBAServiceOnRouter"
                        port="CORBAPortOnRouter"/>
        <ns1:destination service="tns:CORBAService"
                        port="CORBAPort"/>
    </ns1:route>
</definitions>
```

Bypass with Failover Scenario

Bypass routing can be combined with the router failover feature (which can be enabled by setting `multiRoute` to `failover` in the `routing:route` element). In this case, failover support requires cooperation between the CORBA client and the router. Figure 14

gives an overview of a bypass routing with failover scenario.

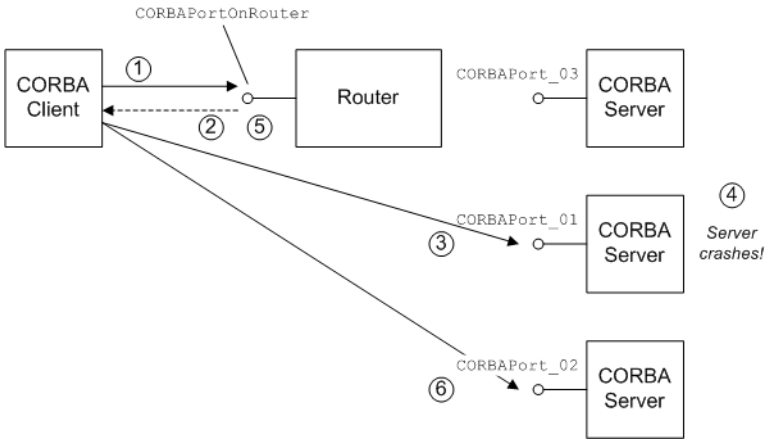

Figure 14: *Bypass Routing with Failover Scenario*

Scenario steps

The bypass routing scenario shown in Figure 14 can be described as follows:

1. The CORBA client sends a GIOP request message to the `CORBAPortOnRouter` endpoint.
2. The router sends a location forward reply, which contains the IOR for one of the destination endpoints in the failover cluster—for example, `CORBAPort_01`.
3. The CORBA client uses the received IOR to open a connection *directly* to the `CORBAPort_01` destination endpoint.
4. If the target server crashes, the CORBA client transparently falls back to the `CORBAPortOnRouter` endpoint.
5. The router again sends a location forward reply, which contains the IOR for another of the destination endpoints in the failover cluster—for example, `CORBAPort_02`.
6. The CORBA client uses the received IOR to open a connection directly to the `CORBAPort_02` destination endpoint.

Configuring bypass with failover

To enable bypass routing with failover, add the following setting to the router's configuration:

```
# Artix Configuration File
bypass_router
{
    plugins:routing:use_bypass = "true";
    ...
};
```

Sample route

Example 7 shows an example of a bypass route with failover enabled. There are three alternative destination endpoints in this failover cluster: `CORBAPort_01`, `CORBAPort_02`, and `CORBAPort_03`.The `multiRoute` attribute must be set to `failover`.

Example 7: *Sample Bypass Route with Failover*

```
<definitions name="" targetNamespace="TargetNamespaceURI"
    ...
    xmlns:tns="TargetNamespaceURI"
    xmlns:ns1="http://schemas.iona.com/routing"
    ...>
    ...
    <ns1:route name="pass_through_route"
               multiRoute="failover">
        <ns1:source      service="tns:CORBAServiceOnRouter"
                         port="CORBAPortOnRouter"/>
        <ns1:destination service="tns:CORBAService_01"
                         port="CORBAPort_01"/>
        <ns1:destination service="tns:CORBAService_02"
                         port="CORBAPort_02"/>
        <ns1:destination service="tns:CORBAService_03"
                         port="CORBAPort_03"/>
    </ns1:route>
</definitions>
```

Bypass with Load Balancing Scenario

Bypass routing can be combined with the router load balancing feature (which can be enabled by setting `multiRoute` to `loadBalance` in the `routing:route` element). When load balancing is combined with bypass routing, the router has the following characteristics:

- Incoming client connections are load-balanced using a round-robin algorithm.
- Load balancing is implemented *per-connection* rather than per-operation. That is, once a client is assigned to a particular destination endpoint, it sends all of its requests to that endpoint.
- Failover is also supported in load balancing scenario. That is, if a server fails, the client is forwarded on to the next healthy server in the cluster (just as in the failover scenario).

Figure 15 gives an overview of a bypass routing with load balancing scenario.

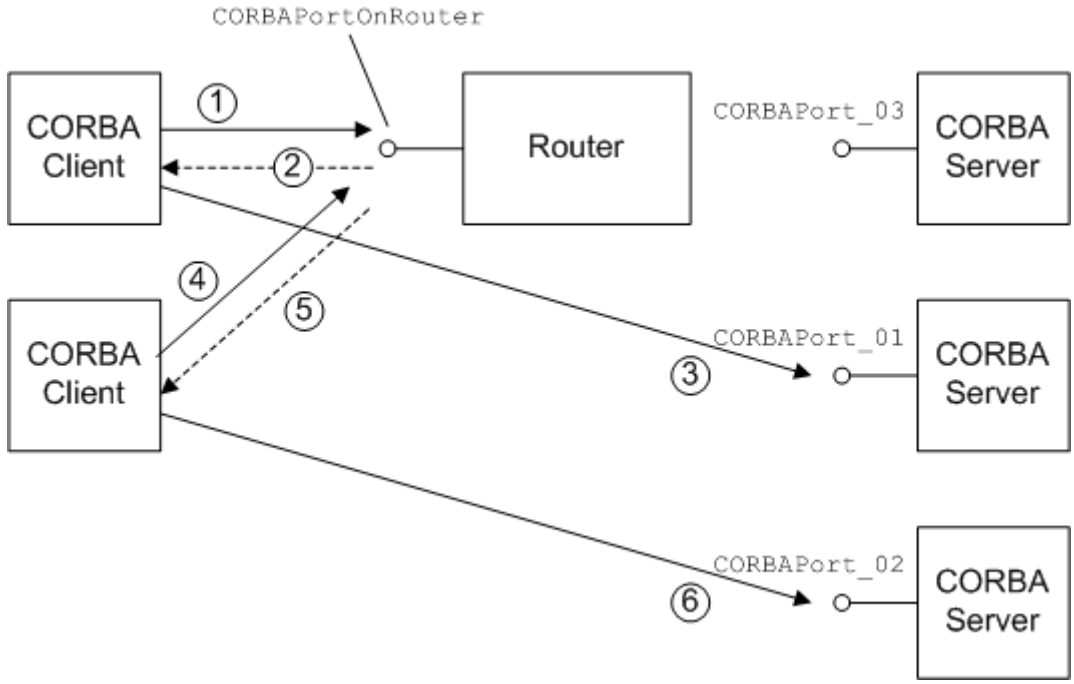

Figure 15: *Bypass Routing with Load Balancing Scenario*

Scenario steps

The bypass routing scenario shown in Figure 15 can be described as follows:

1. The first CORBA client sends a GIOP request message to the `CORBAPortOnRouter` endpoint.
2. The router sends a location forward reply, which contains the IOR for one of the destination endpoints in the load balancing cluster—for example, `CORBAPort_01`.
3. The first CORBA client uses the received IOR to open a connection *directly* to the `CORBAPort_01` destination endpoint.
4. The second CORBA client sends a GIOP request message to the `CORBAPortOnRouter` endpoint.
5. The router sends a location forward reply, which contains the IOR for the next destination endpoint in the load balancing cluster—for example, `CORBAPort_02`. The router load balancing uses a round-robin algorithm to assign destination endpoints to successive clients.
6. The second CORBA client uses the received IOR to open a connection *directly* to the `CORBAPort_02` destination endpoint.

Configuring bypass with load balancing

To enable bypass routing with load balancing, add the following setting to the router's configuration:

```
# Artix Configuration File
bypass_router
{
    plugins:routing:use_bypass = "true";
    ...
};
```

Sample route

Example 8 shows an example of a bypass route with load balancing enabled. There are three alternative destination endpoints in the load balancing cluster: `CORBAPort_01`, `CORBAPort_02`, and `CORBAPort_03`. The `multiRoute` attribute must be set to `loadBalance`.

Example 8: *Sample Bypass Route with Load Balancing*

```
<definitions name="" targetNamespace="TargetNamespaceURI"
    ...
    xmlns:tns="TargetNamespaceURI"
    xmlns:ns1="http://schemas.iona.com/routing"
    ...>
    ...
    <ns1:route name="pass_through_route"
               multiRoute="loadBalance">
        <ns1:source      service="tns:CORBAServiceOnRouter"
                         port="CORBAPortOnRouter"/>
        <ns1:destination service="tns:CORBAService_01"
                         port="CORBAPort_01"/>
        <ns1:destination service="tns:CORBAService_02"
                         port="CORBAPort_02"/>
        <ns1:destination service="tns:CORBAService_03"
                         port="CORBAPort_03"/>
    </ns1:route>
</definitions>
```

Integrating the CORBA Naming Service with Artix

In a mixed Artix/CORBA system, it is often necessary for an Artix application to retrieve an object reference from the CORBA Naming Service. Artix supports a relatively simple configuration option for binding a name to or resolving a name from the CORBA Naming Service: simply set the location attribute of <corba:address> to be a corbaname URL.

How an Artix Client Resolves a Name

Figure 16 shows a typical scenario where an Artix client might need to resolve a name from the CORBA Naming Service. The Artix client, which is configured to have a `corba` binding, connects to a pure CORBA server using the CORBA Naming Service.

To configure the client to resolve the name, you need to specify a `corbaname` URL in the `corba:address` element within a `service`. No programming is required. There are, however, some prerequisites settings in the Artix configuration file that are also required in order to enable the client to find the CORBA Naming Service.

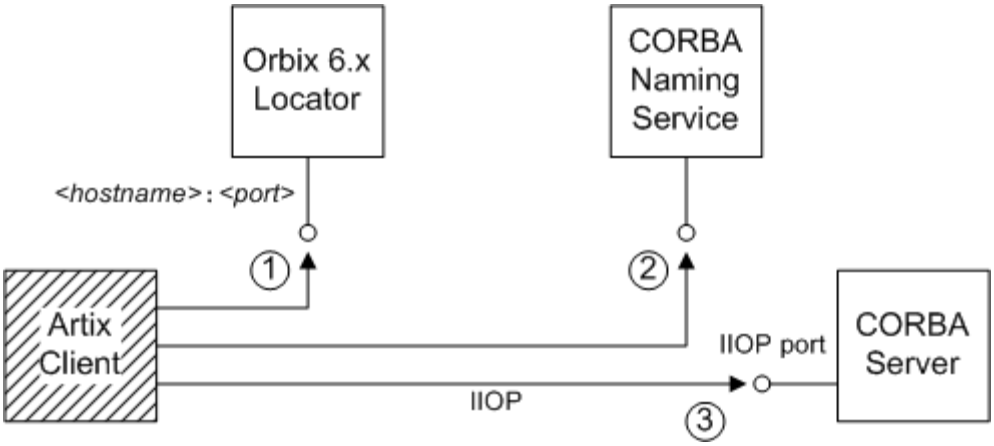

Figure 16: *Artix Client Resolving a Name from the Naming Service*

Resolving steps for Orbix 6.x

Artix performs the following steps to resolve a name in the Orbix 6.x CORBA Naming Service (as shown in Figure 16):

Step	Action
1	The Artix client sends a GIOP *LocateRequest* message to the Orbix locator, whose hostname and port is specified in the Artix configuration file. The *LocateRequest* reply gives the location of the CORBA Naming Service.
2	The Artix client contacts the CORBA Naming Service to resolve the name specified in the WSDL `corba:address` element.
3	The object reference returned from the naming service is used to contact the CORBA server.

Prerequisites

Before configuring the client's WSDL contract to resolve a name from the CORBA Naming Service, you must edit the Artix configuration file to provide some details about the remote naming service.

The configuration settings depend on the kind of ORB you are interoperating with, as follows:

Interoperating with Orbix 6.x, ASP 5.x

In your Artix configuration file (C++ runtime) or Orbix configuration file (Java runtime), add the following lines to the configuration scope used by the Artix client:

```
# Artix Configuration File
artix_client_of_Orbix_6 {
    ...
    initial_references:NameService:reference =
   "corbaloc::<hostname>:<port>/NameService";
    url_resolvers:corbaname:plugin="naming_resolver";
    plugins:naming_resolver:shlib_name="it_naming";
};
```

Where `<hostname>:<port>` is the host and port where the Orbix locator service is running. By default, Orbix 6.x configures the locator `<port>` to be `3075`, but you might need to check the `plugins:locator:iiop:port` setting in your Orbix 6.x configuration file if you are not sure of the value.

Note: The *Orbix locator service* is responsible for keeping track of running Orbix services. It is completely unrelated to the Artix locator service.

Interoperating with Orbix 3.3

In your Artix configuration file (C++ runtime) or Orbix configuration file (Java runtime), add the following lines to the configuration scope used by the Artix client:

```
# Artix Configuration File
artix_client_of_Orbix_33 {
    ...
    initial_references:NameService:reference = "IOR:000000......";

    policies:giop:interop_policy:negotiate_transmission_codeset = "false";
    policies:giop:interop_policy:send_principal = "true";
    policies:giop:interop_policy:send_locate_request = "false";
};
```

The stringified IOR shown in the preceding example, `IOR:000000...`, can be obtained from the 3.3.x Naming Service by starting the NS with the `-I` <*filename*> switch and copying the IOR from the <*filename*> into the configuration file. When using the `IOR:` format, you do not need to load the `naming_resolver` plug-in (the `naming_resolver` is needed only to resolve `corbaloc` URLs).

Interoperating with other ORBs

Generally, the approach used for interoperating with Orbix 3.3 (initializing `initial_references:NameService:reference` with the value of the naming service's IOR) should work for just about any third-party ORB product. You might need to modify some of the GIOP interoperability policies, however. For more details, consult the *Artix Configuration Reference*.

Configure the WSDL service

To configure an Artix client to resolve a name in the CORBA Naming Service, use the `corbaname` URL format in the `<corba:address>` tag, as follows:

```
<service name="CORBAService">
    <port binding="tns:CORBABinding" name="CORBAPort">
        <corba:address location="corbaname:rir:/NameService#StringName"/>
    </port>
</service>
```

Where *StringName* is the name that you want to resolve, specified in the standard CORBA Naming Service string format. For example, if you have a name with `id` equal to `ArtixTest` and `kind` equal to `obj`, contained within a naming context with `id` equal to `Foo` and `kind` equal to `ctx`, the `corbaname` URL would be expressed as:

`corbaname:rir:/NameService#Foo.ctx/ArtixTest.obj`

In other words, the general format of a string name is as follows:

<id>[.<kind>]/<id>[.<kind>]/...

How an Artix Server Binds a Name

Figure 17 shows a typical scenario where an Artix server might need to bind a name to the CORBA Naming Service. In the context of the CORBA Naming Service, *binding a name* means that the server advertises the location of a CORBA object by storing an object reference against a name in the naming service.

To configure the server to bind the name, you need to specify a `corbaname` URL in the `corba:address` element within a `service` (exactly the same configuration as an Artix client). When the Artix server activates the `<service>` or `<port>`, by registering with the Artix Bus, the runtime automatically binds the name in the naming service.

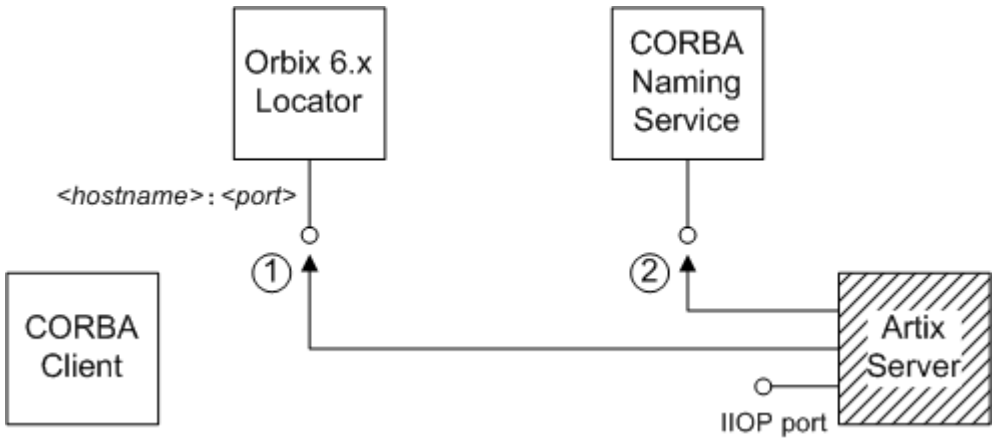

Figure 17: *Artix Server Binding a Name to the Naming Service*

Binding steps for Orbix 6.x

Artix performs the following steps to bind a name in the Orbix 6.x CORBA Naming Service (as shown in Figure 17):

Step	Action
1	The Artix server sends a GIOP *LocateRequest* message to the Orbix locator, whose hostname and port is specified in the Artix configuration file. The *LocateRequest* reply gives the location of the CORBA Naming Service.
2	The Artix server contacts the CORBA Naming Service to bind the name specified in the WSDL `corba:address` element.

Prerequisites

The prerequisites for an Artix server that binds a name to the CORBA Naming Service are identical to the prerequisites for an Artix client that resolves a name—see "Prerequisites" on page 52 for details.

Configure the WSDL service

To configure an Artix server to bind a name in the CORBA Naming Service, use the `corbaname` URL format in the `<corba:address>` tag, as follows:

```
<service name="CORBAService">
    <port binding="tns:CORBABinding" name="CORBAPort">
        <corba:address location="corbaname:rir:/NameService#StringName"/>
    </port>
</service>
```

Where *StringName* is the name that you want to resolve, specified in the standard CORBA Naming Service string format.

This is identical to the configuration for an Artix client, but the server treats this configuration setting differently. When an Artix server activates a service containing a `corbaname` URL, the server automatically binds the given *StringName* into the CORBA naming service.

Binding semantics

The automatic binding performed by an Artix server when it encounters a `corbaname` URL has the following characteristics:

- The binding operation has the semantics of the `CosNaming::NamingContext::rebind()` IDL operation. That is, the bind operation either creates a new binding or clobbers an existing binding of the same name.
- If some of the naming contexts in the *StringName* compound name do not yet exist in the naming service, the Artix server does *not* create the missing contexts.

  For example, if you try to bind a *StringName* with the value `Foo/Bar/SomeName` where neither the `Foo` nor `Foo/Bar` naming contexts exist yet, the Artix server will not bind the given name. You would need to create the naming contexts manually prior to running the Artix server (for example, in Orbix 6.x you could issue the command `itadmin ns newnc` *NameContext*).

Artix Client Integrated with a CORBA Server

This section presents an example scenario of an Artix client integrated with a CORBA server, where the client obtains a CORBA object reference through the CORBA Naming Service.

In summary, the scenario works as follows:

- A CORBA Naming Service from an ORB product (presumed to be Orbix 6.x) is assumed to be running.
- As the CORBA server starts up, it uses the `CosNaming::NamingContext` IDL interface to bind a name to the naming service.
- When the Artix client starts up, the Artix runtime reads the client's WSDL contract, extracts a `corbaname` URL and contacts the naming service to resolve the `corbaname` URL.

CORBA Server Implementation

The code example in this subsection shows you how a server binds a name to the root naming context of the CORBA Naming Service. This shows how a CORBA programmer can use the standard `CosNaming::NamingContext` IDL interface to bind a name.

Note: This is a pure CORBA example; there is no Artix programming involved here.

CORBA server main function

Example 9 shows part of the `main()` function for a CORBA server that registers a name in the CORBA Naming Service. The lines of code shown in bold bind the name, `ArtixTest`, to the root naming context.

Example 9: *CORBA Server that Register a Name in the Naming Service*

```
// C++
...
#include <omg/CosNaming.hh>
...
int main(int argc, char* argv[])
{
    IT_TerminationHandler::set_signal_handler(sig_handler);

    try
    {
        cout << "Initializing the ORB" << endl;
        global_orb = CORBA::ORB_init(argc, argv);
        CORBA::Object_var poa_obj =
            global_orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var root_poa =
            PortableServer::POA::_narrow(poa_obj);
        assert(!CORBA::is_nil(root_poa));

        cout << "Creating objects" << endl;

        HWImplementation hw_servant;
```

Example 9: *CORBA Server that Register a Name in the Naming Service*

```
        PortableServer::ObjectId_var hw_oid =
            root_poa->activate_object(&hw_servant);

        CORBA::Object_var ref=
   root_poa->create_reference_with_id(
                                    hw_oid,
                                    _tc_HelloWorld->id()
                                  );

       // Use the simple NamingContext interface
       CosNaming::NamingContext_var rootContext;

       // Get a reference to the Root Naming Context.
       CORBA::Object_var objVar;
       objVar = global_orb->resolve_initial_references(
                                "NameService"
                           );
       rootContext = CosNaming::NamingContext::_narrow(objVar);

       if (CORBA::is_nil(rootContext.in()))
       {
           cerr << "_narrow returned nil" << endl;
           return 1;
       }

       CosNaming::Name_var tmpName = new CosNaming::Name(1);
       tmpName->length(1);

       tmpName[0].id = CORBA::string_dup("ArtixTest");
       tmpName[0].kind = CORBA::string_dup("");
       rootContext->rebind(tmpName, ref);

        // Activate the POA Manager to allow requests to arrive
        PortableServer::POAManager_var poa_manager =

   root_poa->the_POAManager();
        poa_manager->activate();

        // Give control to the ORB
        //
        global_orb->run();
        return 0;
    }
    catch (CORBA::Exception& e)
    {
        cout << "Error occurred: " << e << endl;
    }
    return 1;
}
```

Demonstration code

If you want to run this CORBA server code in a real example, you could use the following demonstration as a starting point:

ArtixInstallDir/samples/transports/cdr_over_iiop/corba

In the server subdirectory, there is an existing `server.cxx` mainline file that publishes the IOR by saving to a file. To change the server to use the naming service, you can replace the existing server `main()` function with the code shown in Example 9 on page 56.

Note the following points:

- Remember to add the include line, `#include <omg/CosNaming.hh>`, at the start of the `server.cxx` file.
- Edit the server `Makefile`, adding the `it_naming` library to the link list. For example, on Windows you would add `it_naming.lib` to the link list.
- You need a separate ORB product (for example, Orbix) to run the CORBA Naming Service. The Artix product does *not* include a CORBA Naming Service.

Artix Client Configuration

This subsection shows how to configure an Artix client to fetch an object reference from the CORBA Naming Service.

Demonstration configuration

The configuration files referred to in this subsection are taken from the `cdr_over_iiop` demonstration and located in the following directory:

ArtixInstallDir/samples/transports/cdr_over_iiop/etc

The corresponding client application requires no modification. You can choose to run either a C++ version of the client:

`cdr_over_iiop/cxx/client`

Artix configuration file

Example 10 shows the Artix configuration required for the Artix client to interoperate with the Orbix 6.x naming service.

Example 10: *Artix Configuration for Interoperating with Orbix 6 Naming*

```
# Artix Configuration File
include "../../../../../etc/domains/artix.cfg";

demos {
  cdr_over_iiop {
    orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop", "iiop"];

    initial_references:NameService:reference = "corbaloc::localhost:3075/NameService";
    url_resolvers:corbaname:plugin = "naming_resolver";
    plugins:naming_resolver:shlib_name = "it_naming";
```

Example 10: *Artix Configuration for Interoperating with Orbix 6 Naming*

```
    corba {
      orb_plugins = ["iiop_profile", "giop", "iiop"];
    };
  };
};
```

To configure the `cdr_over_iiop` demonstration, edit the `cdr_over_iiop/etc/cdr_over_iiop.cfg` file, inserting the three lines highlighted in bold in Example 10 on page 58. You might need to modify the value of the hostname and port—this example assumes that the Orbix locator service is running on the same host as the client, `localhost`, and listening on the default port, `3075`.

Note: The configuration shown in Example 10 on page 58 is specific to the Orbix 6.x naming service. If you use a different ORB product, you might have to set this configuration differently—see "Prerequisites" on page 52 for more details.

WSDL contract

You also need to edit the client's WSDL contract, specifying the `location` attribute of the `corba:address` element using a `corbaname` URL. Example 11 shows the modifications you need to make to the `corba:address` element in the `cdr_over_iiop/etc/cdr_over_iiop.wsdl` contract file.

Example 11: *CORBA Address Specified as a corbaname URL*

```
<definitions name="cdr_over_iiop" targetNamespace="http://www.iona.com/cdr_over_iiop"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:corba="http://schemas.iona.com/bindings/corba"
    xmlns:corbatm="http://www.iona.com/cdr_over_iiop"
    ... >
    ...
    <service name="HelloWorldService">
        <port binding="corbatm:HelloWorldBinding" name="HelloWorldPort">
           <corba:address location="corbaname:rir:/NameService#ArtixTest"/>
        </port>
    </service>
</definitions>
```

When the client starts up, the Artix runtime automatically retrieves the CORBA object reference by resolving the name, `ArtixTest`, in the scope of the root naming context.

Advanced CORBA Port Configuration

This chapter describes some advanced configuration options for customizing a CORBA port on an Artix server.

Configuring Fixed Ports and Long-Lived IORs

Artix provides a `corba:policy` element that enables you to customize certain CORBA-specific policies for a WSDL service that acts as a CORBA endpoint. Essentially, the `corba:policy` element makes it possible to enable the following features on a CORBA endpoint:

- *Fixed IP port*—the WSDL service listens on the same IP port all the time. This is useful, for example, if the available range of IP ports is restricted or if the service must be accessible through a firewall.
- *Long-lived interoperable object references (IORs)*—the IOR remains valid even after the server is stopped and restarted.

You can configure a WSDL service to behave in one of the following ways:

- Transient service.
- Direct persistent service.

Transient service

By default, a CORBA endpoint is automatically configured to be *transient*. A transient service generates IORs with the following characteristics:

- *Randomly-assigned IP port*—the IP port is assigned by the underlying operating system. Hence, the port is generally different each time the Artix server is run.
- *Short-lived IORs*—the CORBA binding generates IORs in such a way that they are guaranteed to become invalid when the server is stopped and restarted.

Note: In this context, *transient* is a CORBA concept which refers to the `TRANSIENT` value of the `PortableServer::LifespanPolicy`. This notion of transience should *not* be confused with the Artix notion of transience, which is concerned with registering transient servants. The two concepts are completely different.

Direct persistent service

You can optionally configure a CORBA endpoint to be *direct persistent*. A direct persistent service generates IORs with the following characteristics:

- *Fixed IP port*—you can explicitly assign the IP port by configuration. Hence, the IP port remains the same each time the Artix server is run.
- *Long-lived IORs*—the CORBA binding generates IORs in such a way that they remain valid even when the server is stopped and restarted. All of the addressing information embedded in the IOR must remain constant, in particular:
  - *IP port is fixed*—the WSDL service must be configured to listen on a fixed IP port.
  - *POA name is fixed*—the POA name is a CORBA-specific construct that identifies an endpoint.
  - *Object ID in IOR is fixed*—the Object ID is a CORBA-specific construct that identifies a particular object in a given POA instance.
  - *POA is persistent*—a prerequisite for generating long-lived IORs is that the POA must have a life span policy value of `PERSISTENT`.

Configuring a service to be direct persistent

To configure an Artix service to be direct persistent, you must edit both the WSDL file and the Artix configuration file.

Editing the WSDL file

Artix enables you to set direct persistence attributes in WSDL by adding a `corba:policy` element to the WSDL service, as shown in Example 12.

Example 12: *Setting Direct Persistence Attributes in WSDL*

```
<definitions name="" targetNamespace="..."
    ...
    xmlns:corba="CORBANamespace"
    ...>
    ...
    <service name="CORBAServiceName">
        <port binding="tns:CORBABinding" name="CORBAPortName">
            <corba:address location="file:///greeter.ior"/>
            <corba:policy persistent="true"
                          poaname="FQPN"
                          serviceid="ObjectID" />
        </port>
    </service>
</definitions>
```

The `corba:policy` attributes from Example 12 can be explained as follows:

- `xmlns:corba` namespace—the value of the `corba` namespace, *CORBANamespace*.

  ```
  xmlns:corba="http://schemas.iona.com/bindings/corba"
  ```

- `persistent` attribute—by setting this attribute to `true`, you configure the CORBA binding to generate persistent IORs (that is, IORs that continue to be valid even after the Artix server is stopped and restarted). The default value is `false`.

  Note: In CORBA terms, this is equivalent to setting the `PortableServer::LifespanPolicy` policy to `PERSISTENT`.

- `poaname` attribute—in CORBA terminology, a POA is an object that groups CORBA objects together (a kind of container for CORBA objects). It is necessary to set the POA name here, because the POA name is embedded in the generated IORs. The generated IORs would not be long-lived, unless the POA name remains constant. By default, a POA name is automatically generated with the value, {*ServiceNamespace*}*ServiceLocalPart*#*PortName*.

  Note: The POA name, *FQPN*, is a *fully-qualified POA name*. In practice, however, you can only set a simple POA name. Artix currently does not provide a way of creating a POA name hierarchy.

- `serviceid` attribute—in CORBA terminology, this attribute specifies an *Object ID* for a CORBA object. It is necessary to set the Object ID here, because the Object ID is embedded in the server-generated IOR. The Object ID must have a constant value in order for the IOR to be long-lived. By default, the underlying POA would generate a random value for the Object ID.

  Artix currently allows you to set only one Object ID for each port.

  Note: The `serviceid` attribute also implicitly sets the CORBA `PortableServer::IdAssignmentPolicy` policy to `USER_ID`. If the `serviceid` attribute is *not* set, the `PortableServer::IdAssignmentPolicy` policy defaults to `SYSTEM_ID`.

Editing the Artix configuration file

To complete the configuration of direct persistence, you must also set some configuration variables in the relevant scope of the Artix configuration file.

For example, if your Artix server uses the `artix_server` configuration scope, you would add the configuration variables as shown in Example 13.

Example 13: *Setting Direct Persistence Configuration Variables*

```
# Artix Configuration File
...
artix_server {
    ...
    poa:FQPN:direct_persistent="true";
    poa:FQPN:well_known_address="WKA_prefix";
    WKA_prefix:iiop:port="IP_Port";
};
```

The configuration variables from Example 13 can be explained as follows:

- `poa:`*FQPN*`:direct_persistent` variable—you must set this variable to `true`, which configures the CORBA binding to receive *direct* connections from Orbix clients. You should substitute *FQPN* with the POA name from the `poaname` attribute in the WSDL (see Example 12 on page 62).

  Note: In CORBA terms, this is equivalent to setting the `IT_PortableServer::PersistenceModePolicy` policy to `DIRECT_PERSISTENCE`. The alternative policy value, `INDIRECT_PERSISTENCE`, is not compatible with Artix, because it would require connections to be routed through the *Orbix locator service*, which is *not* part of the Artix product.

- `poa:`*FQPN*`:well_known_address` variable—this variable defines a prefix, *WKA_prefix*, which forms part of the variable names that configure a fixed port for the WSDL service. You should substitute *FQPN* with the POA name from the `poaname` attribute in the WSDL.
- *WKA_prefix*`:iiop:port` variable—this variable configures a fixed IP port for the WSDL service associated with *WKA_prefix*.

Fixed port configuration variables

The following IIOP configuration variables can be set for a CORBA endpoint that uses the *WKA_prefix* prefix:

WKA_prefix`:iiop:host = "`*host*`";`

Specifies the hostname, *host*, to publish in the IIOP profile of server-generated IORs. This variable is potentially useful for multi-homed hosts, because it enables you to specify which network card the client should attempt to connect to.

WKA_prefix`:iiop:port = "`*port*`";`

Specifies the fixed IP port, *port*, on which the server listens for incoming IIOP/TLS messages. This port value is also published in the IIOP profile of generated IORs.

WKA_prefix:iiop:listen_addr = "*host*";

Restricts the IIOP/TLS listening point to listen only on the specified address, *host*. It is generally used on multi-homed hosts to limit incoming connections to a particular network interface. The default is to listen on 0.0.0.0 (which represents every network card on the host).

Secure fixed port configuration variables

Additionally, the following secure fixed port configuration variables can be set for a CORBA endpoint that uses the *WKA_prefix* prefix:

WKA_prefix:iiop_tls:host
WKA_prefix:iiop_tls:port
WKA_prefix:iiop_tls:listen_addr

These configuration variables function analogously to their insecure counterparts.

Note: These secure configuration variables will have no effect, unless the iiop_tls plug-in is also loaded. It is strongly recommended that you read the ***Artix Security Guide*** for details of how to configure IIOP/TLS security.

CORBA Timeout Policies

Artix servers that expose a CORBA endpoint can be configured to use CORBA-specific timeout policies. The timeout policies described here affect GIOP transports (for example, the IIOP or IIOP/TLS transports), but do *not* have any affect on non-CORBA transports.

Example

To use the timeout policies, add the relevant configuration variables to the Artix server's configuration scope in the Artix configuration file.

For example, for an Artix server that uses the artix_server configuration scope, you can set the CORBA relative roundtrip timeout as follows:

```
# Artix Configuration File
artix_server {
    # Limit total time for an invocation to 2 seconds
    # (including time for connection and binding establishment).
    policies:relative_roundtrip_timeout = "2000";
}
```

Timeout policies

You can configure the following CORBA timeout policies in your Artix configuration file:

`policies:relative_binding_exclusive_request_timeout`

Limits the amount of time allowed to deliver a request, exclusive of binding attempts. Request delivery is considered complete when the last fragment of the GIOP request is sent over the wire to the target object. This policy's value is set in millisecond units.

`policies:relative_binding_exclusive_roundtrip_timeout`

Limits the amount of time allowed to deliver a request and receive its reply, exclusive of binding attempts. The countdown begins immediately after a binding is obtained for the invocation. This policy's value is set in millisecond units.

`policies:relative_connection_creation_timeout`

Specifies how much time is allowed to resolve each address in an IOR, within each binding iteration. Defaults to 8 seconds.

An IOR can have several `TAG_INTERNET_IOP` (IIOP transport) profiles, each with one or more addresses, while each address can resolve through DNS to multiple IP addresses.

This policy applies to each IP address within an IOR. Each attempt to

resolve an IP address is regarded as a separate attempt to create a

connection. The policy's value is set in millisecond units.

`policies:relative_request_timeout`

Specifies how much time is allowed to deliver a request. Request delivery is considered complete when the last fragment of the GIOP request is sent over the wire to the target object. The timeout-specified period includes any delay in establishing a binding. This policy type is useful to a client that only needs to limit request delivery time. Set this policy's value in millisecond units.

No default is set for this policy; if it is not set, request delivery has unlimited time to complete.

`policies:relative_roundtrip_timeout`

Specifies how much time is allowed to deliver a request and its reply. Set this policy's value in millisecond units. No default is set for this policy; if it is not set, a request has unlimited time to complete.

The timeout countdown begins with the request invocation, and includes the following activities:

- Marshalling in/inout parameters
- Any delay in transparently establishing a binding

If the request times out before the client receives the last fragment of reply data, all received reply data is discarded. In some cases, the client might attempt to cancel the request by sending a GIOP `CancelRequest` message.

Retrying Invocations and Rebinding

Artix lets you configure CORBA policies that customize invocation retries and reconnection. The policies can be grouped into the following categories:

- Retrying invocations.
- Rebinding.

Retrying invocations

The following configuration variables determine how the CORBA binding deals with requests that raise the `CORBA::TRANSIENT` exception with a completion status of `COMPLETED_NO`. In terms of an IIOP connection, a `TRANSIENT` exception is raised if an error occurred before or during an attempt to write to or connect to a socket.

`policies:invocation_retry:backoff_ratio`
: Specifies the degree to which delays between invocation retries increase from one retry to the next. Defaults to 2.

`policies:invocation_retry:initial_retry_delay`
: Specifies the amount of time, in milliseconds, between the first and second retries. Defaults to 100.

> **Note:** The delay between the initial invocation and first retry is always `0`.

`policies:invocation_retry:max_forwards`
: Specifies the number of times an invocation message can be forwarded. Defaults to 20. To specify unlimited forwards, set to -1.

`policies:invocation_retry:max_retries`
: Specifies the number of transparent re-invocations attempted on receipt of a `TRANSIENT` exception. Defaults to 5.

Rebinding

The following configuration variables determine how the CORBA binding deals with requests that raise the `CORBA::COMM_FAILURE` exception with a completion status of `COMPLETED_NO`. In terms of an IIOP connection, a `COMM_FAILURE` exception is raised with a completion status of `COMPLETED_NO`, if the connection went down.

`policies:rebind_policy`
: Specifies the default value for the rebind policy. Can be one of the following:
  - `TRANSPARENT` *(default)*
  - `NO_REBIND`
  - `NO_RECONNECT`

`policies:invocation_retry:max_rebinds`

Specifies the number of transparent rebinds attempted on receipt of a `COMM_FAILURE` exception. Defaults to 5.

Note: This setting is valid only if the effective `policies:rebind_policy` value is `TRANSPARENT`; otherwise, no rebinding occurs.

Artix IDL-to-WSDL Mapping

This chapter describes how the Artix IDL-to-WSDL compiler maps OMG IDL types to WSDL types and how the WSDL types are then mapped to C++.

Introducing CORBA Type Mapping

To ensure that messages are converted into the proper format for a CORBA application to understand, Artix contracts need to unambiguously describe how data is mapped to CORBA data types.

For primitive types, the mapping is straightforward. However, complex types such as structures, arrays, and exceptions require more detailed descriptions.

Unsupported types

The following CORBA types are not supported:

- Value types
- Boxed values
- Local interfaces
- Abstract interfaces
- Forward-declared interfaces

Preprocessor include directives

When converting IDL to WSDL, you can use either of the following preprocessor include directives in your IDL code:

`#include "`*IncludedFile*`"`
`#include <`*IncludedFile*`>`

Both of these include directives are processed in the same way: the preprocessor searches for the specified IDL files in the current include path. The include path consists of the directories specified using the `-I` option on the IDL-to-WSDL compiler command line. For example:

```
idltowsdl -I FirstIncludeDir -I SecondIncludeDir ...
```

IDL Primitive Type Mapping

Mapping chart

Most primitive IDL types are directly mapped to primitive XML Schema types. Table 1 lists the mappings for the supported IDL primitive types.

Table 1: *Primitive Type Mapping for CORBA Plug-in*

IDL Type	XML Schema Type	CORBA Binding Type	Artix C++ Type
`any`	`xsd:anyType`	`corba:any`	`IT_Bus::AnyHolder`
`boolean`	`xsd:boolean`	`corba:boolean`	`IT_Bus::Boolean`
`char`	`xsd:byte`	`corba:char`	`IT_Bus::Byte`
`string`	`xsd:string`	`corba:string`	`IT_Bus::String`
`wchar`	`xsd:string`	`corba:wchar`	`IT_Bus::String`
`wstring`	`xsd:string`	`corba:wstring`	`IT_Bus::String`
`short`	`xsd:short`	`corba:short`	`IT_Bus::Short`
`long`	`xsd:int`	`corba:long`	`IT_Bus::Int`
`long long`	`xsd:long`	`corba:longlong`	`IT_Bus::Long`
`unsigned short`	`xsd:unsignedShort`	`corba:ushort`	`IT_Bus::UShort`
`unsigned long`	`xsd:unsignedInt`	`corba:ulong`	`IT_Bus::UInt`
`unsigned long long`	`xsd:unsignedLong`	`corba:ulonglong`	`IT_Bus::ULong`
`float`	`xsd:float`	`corba:float`	`IT_Bus::Float`
`double`	`xsd:double`	`corba:double`	`IT_Bus::Double`
`long double`	*Not Supported*	*Not Supported*	*Not Supported*
`octet`	`xsd:unsignedByte`	`corba:octet`	`IT_Bus::UByte`
`fixed`	`xsd:decimal`	`corba:fixed`	`IT_Bus::Decimal`
`Object`	`wsa:EndpointReferenceType`	`corba:object`	`WS_Addressing::EndpointReferenceType`
`TimeBase::UtcT`	`xsd:dateTime`[a]	`corba:dateTime`	`IT_Bus::DateTime`

a. The mapping between `xsd:dateTime` and `TimeBase:UtcT` is only partial. For the restrictions see "Unsupported time/date values" on page 71

Unsupported types

Artix does not support the CORBA `long double` type.

Unsupported time/date values

The following `xsd:dateTime` values cannot be mapped to `TimeBase::UtcT`:

- Values with a local time zone. Local time is treated as a 0 UTC time zone offset.
- Values prior to 15 October 1582.
- Values greater than approximately 30,000 A.D.

The following `TimeBase::UtcT` values cannot be mapped to `xsd:dateTime`:

- Values with a non-zero inacclo or inacchi.
- Values with a time zone offset that is not divisible by 30 minutes.
- Values with time zone offsets greater than 14:30 or less than -14:30.
- Values with greater than millisecond accuracy.
- Values with years greater than 9999.

String type

The IDL-to-WSDL mapping for strings is ambiguous, because the `string`, `wchar`, and `wstring` IDL types all map to the same type, `xsd:string`. This ambiguity can be resolved, however, because the generated WSDL records the original IDL type in the CORBA binding description (that is, within the scope of the `<wsdl:binding>` `</wsdl:binding>` tags). Hence, whenever an `xsd:string` is sent over a CORBA binding, it is automatically converted back to the original IDL type (`string`, `wchar`, or `wstring`).

Fixed type

The mapping of `fixed` is a special case. Although `fixed` maps directly to the `xsd:decimal` type, Artix must store additional mapping information in order to support round-trip conversion between WSDL and IDL. Therefore, Artix records the details of the IDL fixed mapping in a `corba:fixed` element (within the scope of the `corba:typeMapping` element). For example, the mapping of a `fixed<6, 2>` type might be recorded as follows:

```
<corba:typeMapping ... >
    <corba:fixed digits="6"
                 scale="2"
                 name="SampleTypes.Money"
                 repositoryID="IDL:SampleTypes/Money:1.0"
                 type="xsd:decimal"/>
</corba:typeMapping>
```

Example

The mapping of primitive types is handled in the CORBA binding section of the Artix contract. For example, consider an input message that has a part, `score`, that is described as an `xsd:int` as shown in Example 14.

Example 14: *WSDL Operation Definition*

```
<message name="runsScored">
  <part name="score"/>
</message>
<portType ...>
  <operation name="getRuns">
    <input message="tns:runsScored" name="runsScored"/>
  </operation>
</portType>
```

It is described in the CORBA binding as shown in Example 15.

Example 15: *Example CORBA Binding*

```
<binding ...>
  <operation name="getRuns">
    <corba:operation name="getRuns">
       <corba:param name="score" mode="in"
   idltype="corba:long"/>
    </corba:operation>
    <input/>
    <output/>
  </operation>
</binding>
```

The IDL is shown in Example 16.

Example 16: *getRuns IDL*

```
// IDL
void getRuns(in score);
```

IDL Complex Type Mapping

This section describes how the complex IDL data types are mapped to WSDL.

This section describes the following data types:

- IDL enum Type
- IDL struct Type
- IDL union Type
- IDL sequence Types
- IDL array Types
- IDL exception Types
- IDL typedef Expressions

IDL enum Type

An IDL enumeration maps to an XML string with enumeration facets. The mapped enumeration is a simple type derived by restriction from the `xsd:string` type.

IDL example

Consider the following definition of an IDL enum type, `SampleTypes::Shape`:

```
// IDL
module SampleTypes {
    enum Shape { Square, Circle, Triangle };
    ...
};
```

WSDL mapping

The IDL-to-WSDL compiler maps the `SampleTypes::Shape` enum to a WSDL restricted simple type, `SampleTypes.Shape`, as follows:

```
<xsd:simpleType name="SampleTypes.Shape">
   <xsd:restriction base="xsd:string">
       <xsd:enumeration value="Square"/>
       <xsd:enumeration value="Circle"/>
       <xsd:enumeration value="Triangle"/>
   </xsd:restriction>
</xsd:simpleType>
```

CORBA type mapping

To support round-trip conversion between WSDL and IDL, Artix records the details of the enumeration type mapping in a `corba:enum` element (within the scope of the `corba:typeMapping` element), as follows:

```
<corba:typeMapping ... >
    <corba:enum name="SampleTypes.Shape"
                repositoryID="IDL:SampleTypes/Shape:1.0"
                type="xsd1:SampleTypes.Shape">
        <corba:enumerator value="Square"/>
        <corba:enumerator value="Circle"/>
        <corba:enumerator value="Triangle"/>
    </corba:enum>
    ...
</corba:typeMapping>
```

C++ mapping

The WSDL-to-C++ compiler maps the `SampleTypes.Shape` type to a C++ class, `SampleTypes_Shape`, as follows:

```
// C++
class SampleTypes_Shape : public IT_Bus::AnySimpleType
{
  public:
    SampleTypes_Shape();
    SampleTypes_Shape(const IT_Bus::String & value);
    ...
    void set_value(const IT_Bus::String & value);
    const IT_Bus::String & get_value() const;
};
```

The value of the enumeration type can be accessed and modified using the `get_value()` and `set_value()` member functions.

IDL struct Type

An IDL structure maps to an `xsd:sequence` type. Each field in the IDL structure maps to an element in the sequence.

IDL example

Consider the following definition of an IDL struct type, `SampleTypes::SampleStruct`:

```
// IDL
module SampleTypes {
    struct SampleStruct {
        string theString;
        long theLong;
    };
};
```

WSDL mapping

The IDL-to-WSDL compiler maps the `SampleTypes::SampleStruct` struct to an XML schema sequence complex type, `SampleTypes.SampleStruct`, as follows:

```
<xsd:complexType name="SampleTypes.SampleStruct">
    <xsd:sequence>
        <xsd:element name="theString" type="xsd:string"/>
        <xsd:element name="theLong" type="xsd:int"/>
    </xsd:sequence>
</xsd:complexType>
```

CORBA type mapping

To support round-trip conversion between WSDL and IDL, Artix records the details of the structure type mapping in a `corba:struct` element (within the scope of the `corba:typeMapping` element), as follows:

```
<corba:typeMapping ... >
    <corba:struct name="SampleTypes.SampleStruct"

   repositoryID="IDL:SampleTypes/SampleStruct:1.0"
                 type="xsd1:SampleTypes.SampleStruct">
        <corba:member idltype="corba:string"
   name="theString"/>
        <corba:member idltype="corba:long"
   name="theLong"/>
    </corba:struct>
</corba:typeMapping>
```

C++ mapping

The WSDL-to-C++ compiler maps the `SampleTypes.SampleStruct` type to a C++ class, `SampleTypes_SampleStruct`, as follows:

```
// C++
class SampleTypes_SampleStruct : public
   IT_Bus::SequenceComplexType
{
  public:
    SampleTypes_SampleStruct();
    SampleTypes_SampleStruct(const
   SampleTypes_SampleStruct& copy);
    ...
    const IT_Bus::String & gettheString() const;
    IT_Bus::String & gettheString();
    void settheString(const IT_Bus::String & val);

    const IT_Bus::Int & gettheLong() const;
    IT_Bus::Int & gettheLong();
    void settheLong(const IT_Bus::Int & val);
};
```

The members of the struct can be accessed and modified using the `get`*`StructMember`*`()` and `set`*`StructMember`*`()` pairs of functions.

IDL union Type

Unions are particularly difficult to describe using the XML schema framework. In the logical data type descriptions, the difficulty is how to describe the union without losing the relationship between the members of the union and the discriminator used to select the members. The easiest method is to describe a union using an `xsd:choice` and list the members in the specified order. The OMG's proposed method is to describe the union as an `xsd:sequence` containing one element for the discriminator and an `xsd:choice` to

describe the members of the union. However, neither of these methods can accurately describe all the possible permutations of a CORBA union.

IDL example

Consider the following definition of an IDL union type, `SampleTypes::Poly`:

```
// IDL
module SampleTypes {
    union Poly switch (short)
    {
      case 0:
        string StringCase0;
      case 1:
      case 2:
        float FloatCase1and2;
      default:
        long caseDef;
    };
};
```

WSDL mapping—default

The IDL-to-WSDL compilers (C++ runtime and Java runtime) generate the following mapping for the IDL union type by default:

```
<complexType name="SampleTypes.Poly">
    <choice>
        <element name="StringCase0" type="string"/>
        <element name="FloatCase1and2" type="float"/>
        <element name="caseDef" type="int"/>
    </choice>
</complexType>
```

In this case, the IDL union maps to `xsd:choice`, where the name of the type is `SampleTypes.Poly`. By default, Artix uses the `xsd:choice` type as the representation of the union throughout the contract.

WSDL mapping—OMG alternative

The IDL-to-WSDL compiler also generates the following *alternative* mapping for the IDL union type:

```
<complexType name="SampleTypes._omg_Poly">
    <sequence>
        <element maxOccurs="1" minOccurs="1"
   name="discriminator"
                        type="short"/>
        <choice maxOccurs="1" minOccurs="0">
            <element name="StringCase0" type="string"/>
            <element name="FloatCase1and2" type="float"/>
            <element name="caseDef" type="int"/>
        </choice>
    </sequence>
</complexType>
```

In this case, the IDL union maps to `xsd:sequence`, where the name of the type is obtained by prepending `_omg_` to the basic type name, giving `SampleTypes._omg_Poly`.

CORBA type mapping

To support round-trip conversion between WSDL and IDL, Artix records the details of the union type mapping in a `corba:union` element (within the scope of the `corba:typeMapping` element), as follows:

```
<corba:typeMapping ... >
    <corba:union discriminator="corba:short"
                 name="SampleTypes.Poly"
                 repositoryID="IDL:SampleTypes/Poly:1.0"
                 type="xsd1:SampleTypes.Poly">
        <corba:unionbranch idltype="corba:string"
                 name="StringCase0">
            <corba:case label="0"/>
        </corba:unionbranch>
        <corba:unionbranch idltype="corba:float"
                 name="FloatCase1and2">
            <corba:case label="1"/>
            <corba:case label="2"/>
        </corba:unionbranch>
        <corba:unionbranch default="true"
                 idltype="corba:long"
                 name="caseDef"/>
    </corba:union>
</corba:typeMapping>
```

C++ mapping

The WSDL-to-C++ compiler maps the `SampleTypes.Poly` type to a C++ class, `SampleTypes_Poly`, as follows:

```
// C++
class SampleTypes_Poly : public IT_Bus::ChoiceComplexType
{
  public:
    ...
    IT_Bus::String & getStringCase0();
    const IT_Bus::String & getStringCase0() const;
    void setStringCase0(const IT_Bus::String & val);

    IT_Bus::Float getFloatCase1and2();
    const IT_Bus::Float getFloatCase1and2() const;
    void setFloatCase1and2(const IT_Bus::Float val);

    IT_Bus::Int getcaseDef();
    const IT_Bus::Int getcaseDef() const;
    void setcaseDef(const IT_Bus::Int val);

    enum PolyDiscriminator
    {
        StringCase0_enum,
        FloatCase1and2_enum,
        caseDef_enum,
        SampleTypes_Poly_MAXLONG=-1
    } m_discriminator;

    PolyDiscriminator get_discriminator() const { ... }
    IT_Bus::UInt get_discriminator_as_uint() const { ... }
    ...
};
```

The value of the union can be modified and accessed using the `get`*UnionMember*`()` and `set`*UnionMember*`()` pairs of functions. The union discriminator can be accessed through the `get_discriminator()` and `get_discriminator_as_uint()` functions.

IDL sequence Types

An IDL sequence maps to a sequence containing a single element that has `minOccurs` equal to zero and `maxOccurs` equal to the sequence's upper bound (`maxOccurs` equals `unbounded`, for an unbounded sequence).

IDL example

Consider the following definition of an IDL unbounded sequence type, `SampleTypes::SeqOfStruct`:

```
// IDL
module SampleTypes {
    typedef sequence< SampleStruct > SeqOfStruct;
    ...
};
```

WSDL mapping

The IDL-to-WSDL compiler maps the `SampleTypes::SeqOfStruct` sequence to a WSDL sequence type with occurrence constraints, `SampleTypes.SeqOfStruct`, as follows:

```
<xsd:complexType name="SampleTypes.SeqOfStruct">
    <xsd:sequence>
        <xsd:element name="item"
            type="xsd1:SampleTypes.SampleStruct"
            minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
```

CORBA type mapping

To support round-trip conversion between WSDL and IDL, Artix records the details of the IDL sequence type mapping in a `corba:sequence` element (within the scope of the `corba:typeMapping` element), as follows:

```
<corba:typeMapping ... >
    <corba:sequence bound="0"
                    elemtype="corbatm:SampleTypes.SampleStruct"
                     name="SampleTypes.SeqOfStruct"

   repositoryID="IDL:SampleTypes/SeqOfStruct:1.0"
                     type="xsd1:SampleTypes.SeqOfStruct"/>
</corba:typeMapping>
```

C++ mapping

The WSDL-to-C++ compiler maps the `SampleTypes.SeqOfStruct` type to a C++ class, `SampleTypes_SeqOfStruct`, as follows:

```
class SampleTypes_SeqOfStruct : public
   IT_Bus::ArrayT<SampleTypes_SampleStruct,
   &SampleTypes_SeqOfStruct_item_qname, 0, -1>
{
  public:
    ...
};
```

The `SampleTypes_SeqOfStruct` class is an Artix C++ array type (based on the `IT_Vector` template). Hence, the array class has an API similar to the `std::vector` type from the C++ Standard Template Library.

Note: IDL bounded sequences map in a similar way to normal IDL sequences, except that the `IT_Bus::ArrayT` base class uses the bounds specified in the IDL.

IDL array Types

An IDL array maps to a sequence containing a single element that sets both `minOccurs` and `maxOccurs` equal to the array bound.

IDL example

Consider the following definition of an IDL union type, `SampleTypes::ArrOfStruct`:

```
// IDL
module SampleTypes {
     typedef SampleStruct ArrOfStruct[10];
    ...
};
```

WSDL mapping

The IDL-to-WSDL compiler maps the `SampleTypes::ArrOfStruct` array to a WSDL sequence type with occurrence constraints, `SampleTypes.ArrOfStruct`, as follows:

```
<xsd:complexType name="SampleTypes.ArrOfStruct">
    <xsd:sequence>
        <xsd:element name="item"
            type="xsd1:SampleTypes.SampleStruct"
            minOccurs="10" maxOccurs="10"/>
    </xsd:sequence>
</xsd:complexType>
```

CORBA type mapping

To support round-trip conversion between WSDL and IDL, Artix records the details of the IDL array type mapping in a `corba:array` element (within the scope of the `corba:typeMapping` element), as follows:

```
<corba:typeMapping ... >
    <corba:array bound="10"
                 elemtype="corbatm:SampleTypes.SampleStruct"
                 name="SampleTypes.ArrOfStruct"
                repositoryID="IDL:SampleTypes/ArrOfStruct:1.0"
                 type="xsd1:SampleTypes.ArrOfStruct"/>
</corba:typeMapping>
```

C++ mapping

The WSDL-to-C++ compiler maps the `SampleTypes.ArrOfStruct` type to a C++ class, `SampleTypes_ArrOfStruct`, as follows:

```
class SampleTypes_ArrOfStruct : public
   IT_Bus::ArrayT<SampleTypes_SampleStruct,
   &SampleTypes_ArrOfStruct_item_qname, 10, 10>
{
    ...
};
```

The `SampleTypes_ArrOfStruct` class is an Artix C++ array type (based on the `IT_Vector` template). The array class has an API similar to the `std::vector` type from the C++ Standard Template Library, except that the size of the vector is restricted to the specified array length, `10`.

IDL exception Types

An IDL exception type maps to an `xsd:sequence` type and to an exception message. Each field in the IDL exception maps to an element in the `xsd:sequence`.

IDL example

Consider the following definition of an IDL exception type, `SampleTypes::GenericException`:

```
// IDL
module SampleTypes {
    exception GenericExc {
        string reason;
    };
    ...
};
```

WSDL mapping

The C++ runtime version of the IDL-to-WSDL compiler maps the `SampleTypes::GenericExc` exception to a WSDL sequence type, `SampleTypes.GenericExc`, and to a WSDL fault message, `SampleTypes.GenericExc`, as follows:

```
<xsd:complexType name="SampleTypes.GenericExc">
    <xsd:sequence>
        <xsd:element name="reason" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
...
<xsd:element name="SampleTypes.GenericExc"
    type="xsd1:SampleTypes.GenericExc"/>
...
<message name="SampleTypes.GenericExc">
    <part element="xsd1:SampleTypes.GenericExc"
          name="exception"/>
</message>
```

The output from the Java runtime version of the IDL-to-WSDL compiler is slightly different. The WSDL sequence type is named `SampleTypes.GenericExcType` instead of `SampleTypes.GenericExc`. For example:

```
<xsd:complexType name="SampleTypes.GenericExcType">
    <xsd:sequence>
        <xsd:element name="reason" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
...
<xsd:element name="SampleTypes.GenericExc"
    type="xsd1:SampleTypes.GenericExcType"/>
...
<message name="SampleTypes.GenericExc">
    <part element="xsd1:SampleTypes.GenericExc"
          name="exception"/>
</message>
```

CORBA type mapping

To support round-trip conversion between WSDL and IDL, Artix records the details of the IDL exception type mapping in a `corba:exception` element (within the scope of the `corba:typeMapping` element).

For example, the C++ runtime version of the IDL-to-WSDL compiler generates the following `corba:exception` element:

```
<corba:typeMapping ... >
    <corba:exception name="SampleTypes.GenericExc"

   repositoryID="IDL:SampleTypes/GenericExc:1.0"
                      type="xsd1:SampleTypes.GenericExc">
        <corba:member idltype="corba:string"
   name="reason"/>
    </corba:exception>
</corba:typeMapping>
```

C++ mapping

The WSDL-to-C++ compiler maps the `SampleTypes.GenericExc` type and `SampleTypes.GenericExc` message type to the C++ classes, `SampleTypes_GenericExc` and `SampleTypes_GenericExcException`, as follows:

```
// C++
class SampleTypes_GenericExc : public
   IT_Bus::SequenceComplexType
{
  public:
    SampleTypes_GenericExc();
    ...
    const IT_Bus::String & getreason() const;
    IT_Bus::String & getreason();
    void setreason(const IT_Bus::String & val);
};
...
class SampleTypes_GenericExcException
  : public IT_Bus::UserFaultException,
    public
   IT_Bus::Rethrowable<SampleTypes_GenericExcException>
{
  public:
    SampleTypes_GenericExcException();
    ...
    const SampleTypes_GenericExc & getexception() const;
    SampleTypes_GenericExc & getexception();
    void setexception(const SampleTypes_GenericExc & val);
    ...
};
```

IDL typedef Expressions

If a type is aliased in IDL, using a `typedef` expression, Artix simply replaces the type alias with the original type when mapping to WSDL.

Note: The typedef that defines an IDL sequence or an IDL array is treated as a special case, with a specific C++ class being generated to represent the sequence or array type.

IDL example

Consider the following IDL typedef that defines an alias of a `float`, `SampleTypes::FloatAlias`, and an alias of a `struct`, `SampleTypes::SampleStruct`:

```
// IDL
module SampleTypes {
    typedef float FloatAlias;
    typedef SampleStruct SampleStructAlias;
    ...
};
```

CORBA type mapping

To support round-trip conversion between WSDL and IDL, Artix records the details of each IDL alias mapping in a `corba:alias` element (within the scope of the `corba:typeMapping` element), as follows:

```
<corba:typeMapping ... >
    <corba:alias basetype="corba:float"
            name="SampleTypes.FloatAlias"
            repositoryID="IDL:SampleTypes/FloatAlias:1.0"
            type="xsd:float"/>
    <corba:alias
   basetype="corbatm:SampleTypes.SampleStruct"
            name="SampleTypes.SampleStructAlias"

   repositoryID="IDL:SampleTypes/SampleStructAlias:1.0"
            type="xsd1:SampleTypes.SampleStruct"/>
</corba:typeMapping>
```

WSDL mapping

The IDL-to-WSDL compiler maps the `SampleTypes::FloatAlias` type alias directly to the type, `xsd:float` and the `SampleTypes::SampleStructAlias` type alias directly to the type, `SampleTypes.SampleStruct`.

IDL Module and Interface Mapping

This section describes the mapping of IDL modules and interfaces.

Mapping IDL Modules and Interfaces to C++

This section describes the Artix C++ mapping for the following IDL constructs:

- Module mapping.
- Interface mapping.
- Object reference mapping.
- Operation mapping.
- Attribute mapping.

Module mapping

An IDL identifier appearing within the scope of an IDL module, *ModuleName*::*Identifier*, maps to a C++ identifier of the form *ModuleName_Identifier*. That is, the IDL scoping operator, `::`, maps to an underscore, _, in C++.

Although IDL modules do *not* map to namespaces under the Artix C++ mapping, it is possible nevertheless to put generated C++ code into a namespace using the `-n` switch to the WSDL-to-C++ compiler.

For example, if you pass a namespace, `TEST`, to the WSDL-to-C++ `-n` switch, the *ModuleName*::*Identifier* IDL identifier would map to `TEST`::*ModuleName_Identifier*.

Interface mapping

An IDL interface, *InterfaceName*, maps to a C++ class of the same name, *InterfaceName*. If the interface is defined in the scope of a module, that is *ModuleName*::*InterfaceName*, the interface maps to the *ModuleName_InterfaceName* C++ class.

If an IDL data type, *TypeName*, is defined within the scope of an IDL interface, that is *ModuleName*::*InterfaceName*::*TypeName*, the type maps to the *ModuleName_InterfaceName_TypeName* C++ class.

Object reference mapping

When an IDL interface is used as an operation parameter or return type, it is mapped to the `WS_Addressing::EndpointReferenceType` C++ type.

For example, consider an operation, `get_foo()`, that returns a reference to a `Foo` interface as follows:

```
// IDL
interface Foo {};

interface Bar {
    Foo get_foo();
};
```

The `get_foo()` IDL operation then maps to the following C++ function:

```
// C++
void get_foo(
    WS_Addressing::EndpointReferenceType & var_return
) IT_THROW_DECL((IT_Bus::Exception));
```

Note that this mapping is qualitatively different from the OMG IDL-to-C++ mapping. In the Artix mapping, the `get_foo()` operation does not return a pointer to a `Foo` proxy object. Instead, you must construct the `Foo` proxy object in a separate step, by passing the `WS_Addressing::EndpointReferenceType` object into the `FooClient` constructor.

Nil object reference

A CORBA nil object reference maps to an empty endpoint reference. Conventionally, the address of an empty endpoint reference is represented by the following URI:

`http://www.w3.org/2005/08/addressing/none`

Operation mapping

Example 17 shows two IDL operations defined within the `SampleTypes::Foo` interface. The first operation is a regular IDL operation, `test_op()`, and the second operation is a oneway operation, `test_oneway()`.

Example 17: *Example IDL Operations*

```
// IDL
module SampleTypes {
    ...
    interface Foo {
        ...
        SampleStruct test_op(
            in SampleStruct    in_struct,
            inout SampleStruct inout_struct,
            out SampleStruct   out_struct
        ) raises (GenericExc);

        oneway void test_oneway(in string in_str);
    };
};
```

The operations from the preceding IDL, Example 17 on page 87, map to C++ as shown in Example 18,

Example 18: *Mapping IDL Operations to C++*

```
// C++
class SampleTypes_Foo
{
  public:
    ...
1   virtual void test_op(
        const TEST::SampleTypes_SampleStruct & in_struct,
        TEST::SampleTypes_SampleStruct & inout_struct,
        TEST::SampleTypes_SampleStruct & var_return,
        TEST::SampleTypes_SampleStruct & out_struct
    ) IT_THROW_DECL((IT_Bus::Exception)) = 0;

2   virtual void test_oneway(
        const IT_Bus::String & in_str
    ) IT_THROW_DECL((IT_Bus::Exception)) = 0;
};
```

The preceding C++ operation signatures can be explained as follows:

1. The C++ mapping of an IDL operation always has the return type `void`. If a return value is defined in IDL, it is mapped as an out parameter, `var_return`.

   The order of parameters in the C++ function signature, `test_op()`, is determined as follows:

   - First, the in and inout parameters appear in the same order as in IDL, ignoring the out parameters.
   - Next, the return value appears as the parameter, `var_return` (with the same semantics as an out parameter).

- Finally, the out parameters appear in the same order as in IDL, ignoring the in and inout parameters.

2. The C++ mapping of an IDL oneway operation is straightforward, because a oneway operation can have only `in` parameters and a `void` return type.

Attribute mapping

Example 19 shows two IDL attributes defined within the `SampleTypes::Foo` interface. The first attribute is readable and writable, `str_attr`, and the second attribute is read only, `struct_attr`.

Example 19: *Example IDL Attributes*

```
// IDL
module SampleTypes {
    ...
    interface Foo {
        ...
        attribute string                str_attr;
        readonly attribute SampleStruct struct_attr;
    };
};
```

The attributes from the preceding IDL, Example 19 on page 88, map to C++ as shown in Example 20,

Example 20: *Mapping IDL Attributes to C++*

```
// C++
class SampleTypes_Foo
{
  public:
    ...
1   virtual void _get_str_attr(
        IT_Bus::String & var_return
    ) IT_THROW_DECL((IT_Bus::Exception)) = 0;

    virtual void _set_str_attr(
        const IT_Bus::String & _arg
    ) IT_THROW_DECL((IT_Bus::Exception)) = 0;

2   virtual void _get_struct_attr(
        TEST::SampleTypes_SampleStruct & var_return
    ) IT_THROW_DECL((IT_Bus::Exception)) = 0;
};
```

The preceding C++ attribute signatures can be explained as follows:

1. A normal IDL attribute, *AttributeName*, maps to a pair of accessor and modifier functions in C++, `_get_`*AttributeName*`()`, `_set_`*AttributeName*`()`.
2. An IDL read-only attribute, *AttributeName*, maps to a single accessor function in C++, `_get_`*AttributeName*`()`.

Artix WSDL-to-IDL Mapping

This chapter describes how the Artix WSDL-to-IDL compiler maps WSDL types to OMG IDL types.

This chapter discusses the following topics:

Simple Types

This section describes the mapping of simple WSDL types to IDL. It deals with:

Atomic Types

soapenc atomic types

Artix maps the `soapenc:string` type to the `string` IDL type (where the `soapenc` namespace prefix is identified with the `http://schemas.xmlsoap.org/soap/encoding/` namespace).

Table of XSD atomic types

Table 2 shows how the XSD schema atomic types map to IDL.

Table 2: *XSD Schema Simple Types Mapping to IDL*

XSD Schema Type	IDL Type
xsd:boolean	boolean
xsd:byte	char
xsd:unsignedByte	octet
xsd:short	short
xsd:unsignedShort	unsigned short
xsd:int	long
xsd:unsignedInt	unsigned long
xsd:long	long long
xsd:unsignedLong	unsigned long long
xsd:float	float
xsd:double	double
xsd:string	string
xsd:normalizedString	string
xsd:token	string
xsd:language	string
xsd:NMTOKEN	string
xsd:NMTOKENS	*Not supported*
xsd:Name	string
xsd:NCName	string
xsd:ID	string
xsd:QName	string
xsd:dateTime	TimeBase::UtcT
xsd:date	string
xsd:time	string
xsd:gDay	string
xsd:gMonth	string
xsd:gMonthDay	string
xsd:gYear	string
xsd:gYearMonth	string

Table 2: *XSD Schema Simple Types Mapping to IDL*

XSD Schema Type	IDL Type
xsd:duration	string
xsd:decimal	*Typedef of* fixed<31,6>
xsd:integer	long long
xsd:positiveInteger	unsigned long long
xsd:negativeInteger	long long
xsd:nonPositiveInteger	long long
xsd:nonNegativeInteger	unsigned long long
xsd:base64Binary	base64BinarySeq *(typedef of* sequence<octet>*)*
xsd:hexBinary	hexBinarySeq *(typedef of* sequence<octet>*)*
soapenc:base64	base64Seq *(typedef of* sequence<octet>*)*
xsd:ID	*Not supported.*

String Type

Artix can map strings both from the soapenc schema and from the XSD schema, as follows:

- soapenc string type.
- XSD string type.

soapenc string type

Artix maps the soapenc:string type to the string IDL type (where the soapenc namespace prefix is identified with the http://schemas.xmlsoap.org/soap/encoding/ namespace).

XSD string type

By default, xsd:string maps to the ordinary IDL string type.

If you are planning to use international strings, however, you might want xsd:string to map to the IDL wide string type, wstring, instead. The wsdltocorba utility does not provide an option to change the default mapping, but you can easily alter the mapping by manually editing the contents of the CORBA <binding> tag in the WSDL.

Default CORBA binding

Consider, for example, how to add a CORBA binding to the `Greeter` port type (see the `hello_world.wsdl` file located in *ArtixInstallDir*`/samples/basic/hello_world_soap_http/etc`). You can add a CORBA binding by entering the following command:

```
> wsdltocorba -corba -i Greeter hello_world.wsdl
```

The WSDL output from this command, `hello_world-corba.wsdl`, includes a new CORBA binding, `GreeterCORBABinding`, as shown in Example 21. The contents of this `binding` element essentially determine the WSDL-to-CORBA mapping for the port type. Some parameters and return types in the binding are declared to have an `idltype` attribute of `corba:string`, which means they map to the IDL `string` type.

Example 21: *Default CORBA Binding Generated by wsdltocorba*

```
<definitions ... >
    ...
    <binding name="GreeterCORBABinding" type="tns:Greeter">
        <corba:binding repositoryID="IDL:Greeter:1.0"/>
        <operation name="sayHi">
            <corba:operation name="sayHi">
                <corba:return idltype="corba:string" name="theResponse"/>
            </corba:operation>
            <input name="sayHiRequest"/>
            <output name="sayHiResponse"/>
        </operation>
        <operation name="greetMe">
            <corba:operation name="greetMe">
                <corba:param idltype="corba:string" mode="in" name="me"/>
                <corba:return idltype="corba:string" name="theResponse"/>
            </corba:operation>
            <input name="greetMeRequest"/>
            <output name="greetMeResponse"/>
        </operation>
    </binding>
</definitions>
```

Manually modified CORBA binding

To alter the WSDL-to-IDL string mapping, replace some or all of the instances of `corba:string` by `corba:wstring`. Example 22 shows the result of replacing all instances of `corba:string` by `corba:wstring`.

Example 22: *Manually Modified CORBA Binding*

```
<definitions ... >
    ...
    <binding name="GreeterCORBABinding" type="tns:Greeter">
        <corba:binding repositoryID="IDL:Greeter:1.0"/>
        <operation name="sayHi">
            <corba:operation name="sayHi">
```

Example 22: *Manually Modified CORBA Binding*

```
                <corba:return idltype="corba:wstring"
  name="theResponse"/>
            </corba:operation>
            <input name="sayHiRequest"/>
            <output name="sayHiResponse"/>
        </operation>
        <operation name="greetMe">
            <corba:operation name="greetMe">
                <corba:param idltype="corba:wstring" mode="in"
  name="me"/>
                <corba:return idltype="corba:wstring"
  name="theResponse"/>
            </corba:operation>
            <input name="greetMeRequest"/>
            <output name="greetMeResponse"/>
        </operation>
    </binding>
</definitions>
```

Generated IDL

Example 23 shows the IDL that would be generated from the modified CORBA binding in Example 22 on page 92.

Example 23: *IDL Generated from the Modified CORBA Binding*

```
// IDL

interface Greeter {
    wstring sayHi();
    wstring greetMe(in wstring me);
};
```

To generate this IDL interface, you would enter the following command:

```
> wsdltocorba -idl -b GreeterCORBABinding
   hello_world-corba.wsdl
```

Date and Time Types

The WSDL-to-IDL compiler maps the `xsd:dateTime` type to the `TimeBase::UtcT` IDL type.

Note: The mapping is subject to certain restrictions, as detailed below.

TimeBase::UtcT type

The `TimeBase::UtcT` type, which holds a UTC time value, is defined in the OMG's *CORBA Time Service* specification. Example 24 shows the definition of `UtcT` in the `TimeBase` module.

Example 24: *Definition of the TimeBase IDL Module*

```
// IDL
module TimeBase
{
    typedef unsigned long long TimeT;
    typedef TimeT              InaccuracyT;
    typedef short              TdfT;

    struct UtcT
    {
        TimeT          time;
        unsigned long  inacclo;
        unsigned short inacchi;
        TdfT           tdf;
    };

    struct IntervalT
    {
        TimeT lower_bound;
        TimeT upper_bound;
    };
};
```

Unsupported time/date values

The following `xsd:dateTime` values cannot be mapped to `TimeBase::UtcT`:

- Values with a local time zone. Local time is treated as a 0 UTC time zone offset.
- Values prior to 15 October 1582.
- Values greater than approximately 30,000 A.D.

The following `TimeBase::UtcT` values cannot be mapped to `xsd:dateTime`:

- Values with a non-zero `inacclo` or `inacchi`.
- Values with a time zone offset that is not divisible by 30 minutes.
- Values with time zone offsets greater than 14:30 or less than -14:30.
- Values with greater than millisecond accuracy.
- Values with years greater than 9999.

Duration Type

The WSDL-to-IDL compiler maps the `xsd:duration` type to the `string` IDL type.

A *duration* represents an interval of time measured in years, months, days, hours, minutes, and seconds. This type is needed for representing the sort of time intervals that commonly appear in business and legal documents.

Lexical representation

The lexical representation of a positive time duration is as follows:

P*<years>*Y*<months>*M*<days>*DT*<hours>*H*<minutes>*M*<seconds>*S

Where *<years>*, *<months>*, *<days>*, *<hours>*, and *<minutes>* are non-negative integers and *<seconds>* is a non-negative decimal. The *<seconds>* field can have an arbitrary number of decimal digits, but Artix considers the digits only up to millisecond precision. The `P`, `Y`, `M`, `D`, `T`, `H`, `M`, and `S` separator characters must all be upper case. The `T` is the date/time separator. To represent a negative time duration, you can add a minus sign, `-`, in front of the `P` character.

Here are some examples:

```
P2Y6M10DT12H20M15S
-P1Y0M0DT0H0M0.001S
```

You can abbreviate the duration string by omitting any fields that are equal to zero. You must omit the date/time separator, `T`, if and only if all of the time fields are absent. For example, `P1Y` would represent one year.

Deriving Simple Types by Restriction

Most derived simple types are mapped as if they had been declared to be the base type. For example, XSD types derived from `xsd:string` are treated as if they were declared as `xsd:string` and are therefore mapped to the IDL `string` type.

Exceptionally, derived simple types declared using the `<enumeration>` facet are treated as a special case: enumerated simple types are mapped to an IDL `enum` type.

Unchecked facets

The following facets can be used, but are not checked at runtime:

- `length`
- `minLength`
- `maxLength`
- `pattern`
- `enumeration`
- `whiteSpace`
- `maxInclusive`
- `maxExclusive`

- minInclusive
- minExclusive
- totalDigits
- fractionDigits

Checked facets

The following facets are supported and checked at runtime:

- enumeration

Example with a maxLength facet

The following example shows how you can use the <maxLength> facet to define a string whose length is limited to 100 characters:

```
<xsd:simpleType name="String100">
    <xsd:restriction base="xsd:string">
        <xsd:maxLength value="100"/>
    </xsd:restriction>
</xsd:simpleType>
```

The WSDL-to-IDL mapping maps this String100 type to the string type.

Example with enumeration facets

The following example shows how to define an enumerated type, ColorEnum, using the <enumeration> facet:

```
<xsd:simpleType name="ColorEnum">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="RED"/>
        <xsd:enumeration value="GREEN"/>
        <xsd:enumeration value="BLUE"/>
    </xsd:restriction>
</xsd:simpleType>
```

The WSDL-to-IDL mapping maps this ColorEnum type to the following IDL enum type.

```
// IDL
enum ColorEnum {
    RED,
    GREEN,
    BLUE
};
```

List Type

An `xsd:list` type maps to an IDL sequence type, `sequence<`*MappedElementType*`>`, where *MappedElementType* is the IDL type representing the list elements.

There are two styles of list declaration, both of which are supported in Artix:

- Lists defined using itemType.
- Lists defined by derivation.

Lists defined using itemType

Where the list element type is a schema atomic type, you can define the list type using the `itemType` attribute. For example, a list of strings can be defined as follows:

```
<xsd:simpleType name="StringList">
    <xsd:list itemType="xsd:string"/>
</xsd:simpleType>
```

This maps to the following IDL type:

```
// IDL
typedef sequence<string> StringList;
```

Lists defined by derivation

Where the list element type is derived from a schema atomic type (by the application of various restricting facets), you can define the list type using a `restriction` element. For example, you can define a list of restricted integers as follows:

```
<xsd:simpleType name="IntList">
    <xsd:list>
        <xsd:simpleType>
            <xsd:restriction base="xsd:int">
                <xsd:maxInclusive value="1000"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:list>
</xsd:simpleType>
```

This maps to the following IDL type:

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

Unsupported Simple Types

This subsection lists the XSD simple types that are not supported by the `wsdltocorba` mapping utilities.

Unsupported types

The following XSD simple types are not supported by the WSDL-to-IDL mapping:

```
xsd:ENTITY
xsd:ENTITIES
xsd:IDREF
xsd:IDREFS
xsd:NMTOKENS
xsd:NOTATION
xsd:union
```

Complex Types

This section describes the mapping of complex WSDL types to IDL. It deals with the following types:

- Sequence Complex Types
- Choice Complex Types
- All Complex Types
- Attributes
- Nesting Complex Types
- Deriving a Complex Type from a Simple Type
- Deriving a Complex Type from a Complex Type
- Arrays

Sequence Complex Types

The XSD sequence complex type maps to an IDL `struct` type, where each element of the original sequence maps to a member of the IDL `struct`.

Occurrence constraints

The WSDL-to-IDL mapping does *not* support occurrence constraints on the `sequence` element. If `minOccurs` or `maxOccurs` attribute settings appear in the `sequence` element, they are ignored by the WSDL-to-IDL compiler.

On the other hand, elements appearing *within* the `sequence` element can define occurrence constraints—see "Arrays" on page 107.

WSDL example

Example 25 shows an XSD sequence type with three simple elements.

Example 25: *Definition of a Sequence Complex Type in WSDL*

```
<xsd:complexType name="SimpleStruct">
    <xsd:sequence>
        <xsd:element name="varFloat" type="xsd:float"/>
        <xsd:element name="varInt" type="xsd:int"/>
        <xsd:element name="varString" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

IDL mapping

Example 26 shows the result of mapping the `SimpleStruct` type (from the preceding Example 25) to IDL.

Example 26: *Mapping of SimpleStruct to IDL*

```
// IDL
struct SimpleStruct {
    float varFloat;
    long varInt;
    string varString;
};
```

Choice Complex Types

The XSD choice complex type maps to an IDL `union` type, where each element of the original choice maps to a member of the IDL `union`.

Occurrence constraints

Artix does not support occurrence constraints on the `choice` element.

WSDL example

Example 27 shows an XSD choice type with three elements.

Example 27: *Definition of a Choice Complex Type in WSDL*

```
<xsd:complexType name="SimpleChoice">
    <xsd:choice>
        <xsd:element name="varFloat" type="xsd:float"/>
        <xsd:element name="varInt" type="xsd:int"/>
        <xsd:element name="varString" type="xsd:string"/>
    </xsd:choice>
</xsd:complexType>
```

IDL mapping

Example 28 shows the result of mapping the `SimpleChoice` type (from the preceding Example 27) to IDL.

Example 28: *Mapping of SimpleChoice to IDL*

```
// IDL
union SimpleChoice switch (long) {
    case 0:
        float varFloat;
    case 1:
        long varInt;
    case 2:
        string varString;
};
```

All Complex Types

The XSD all complex type maps to an IDL `struct` type, where each element of the original all maps to a member of the IDL `struct`.

Occurrence constraints

Artix does not support occurrence constraints on the `all` element.

WSDL example

Example 29 shows an XSD all type with three simple elements.

Example 29: *Definition of an All Complex Type in WSDL*

```
<xsd:complexType name="SimpleAll">
    <xsd:all>
        <xsd:element name="varFloat" type="xsd:float"/>
        <xsd:element name="varInt" type="xsd:int"/>
        <xsd:element name="varString" type="xsd:string"/>
    </xsd:all>
</xsd:complexType>
```

IDL mapping

Example 30 shows the result of mapping the `SimpleAll` type (from the preceding Example 29) to IDL.

Example 30: *Mapping of SimpleAll to IDL*

```
// IDL
struct SimpleAll {
    float varFloat;
    long varInt;
    string varString;
};
```

Attributes

Attributes of a sequence type or of an all type map to additional members of an IDL `struct`. The type representing an attribute in IDL is defined as a *nillable type* (see "Nillable Types" on page 112 for details). This makes it possible for attributes to be treated as optional.

Attributes can be declared within the scope of the `xsd:complexType` element. Hence, you can include attributes in the definitions of an all type, a sequence type, and a choice type.

Note: Attributes of a choice type are currently *not* supported by the WSDL-to-IDL mapping.

The declaration of an attribute in a complex type has the following syntax:

```
<xsd:complexType name="TypeName">
    <xsd:attribute name="AttrName" type="AttrType"
                   use="[optional|required|prohibited]"/>
    ...
</xsd:complexType>
```

Attribute use

The `use` attribute setting is ignored by the WSDL-to-IDL mapping.

Because attributes are declared as nillable types in IDL, the attributes are effectively `optional` by default. If the attribute `use` is defined as `required` or `prohibited`, however, it is up to the developer to enforce these conditions.

WSDL example

Example 31 shows an XSD sequence type, which is declared to have two attributes, `varAttrString` and `varAttrIntOptional`.

Example 31: *Definition of a Complex Type with Attributes in WSDL*

```
<xsd:complexType name="SimpleStructWithAttributes">
    <xsd:sequence>
        <xsd:element name="varFloat" type="xsd:float"/>
        <xsd:element name="varInt" type="xsd:int"/>
        <xsd:element name="varString" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="varAttrString" type="xsd:string"/>
    <xsd:attribute name="varAttrIntOptional" type="xsd:int"
                   use="optional"/>
</xsd:complexType>
```

IDL mapping

Example 32 shows the result of mapping the `SimpleStructWithAttributes` type (from the preceding Example 31) to IDL.

Example 32: *Mapping of SimpleStructWithAttributes to IDL*

```
// IDL
union string_nil switch(boolean) {
    case TRUE:
        string value;
};
union long_nil switch(boolean) {
    case TRUE:
        long value;
};

struct SimpleStructWithAttributes {
    string_nil varAttrString;
    long_nil varAttrIntOptional;
    float varFloat;
    long varInt;
    string varString;
};
```

Nesting Complex Types

It is possible to nest complex types within each other. When mapped to IDL, the nested complex types map to a nested hierarchy of structs, where each instance of a nested type is declared as a member of another struct.

Avoiding anonymous types

In general, it is recommended that you name types that are nested inside other types, instead of using anonymous types. This results in simpler code when the types are mapped to IDL.

Note: The WSDL-to-IDL mapping has only limited support for mapping anonymous type, which does not work in all cases.

WSDL example

Example 33 shows the definition of a nested sequence type, `NestedStruct`, which contains another sequence type, `SimpleStruct`, as an element.

Example 33: *Definition of a Nested Type in WSDL*

```
<xsd:complexType name="SimpleStruct">
    <xsd:sequence>
        <xsd:element name="varFloat" type="xsd:float"/>
        <xsd:element name="varInt" type="xsd:int"/>
        <xsd:element name="varString" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="NestedStruct">
    <xsd:sequence>
        <xsd:element name="varString" type="xsd:string"/>
        <xsd:element name="varInt" type="xsd:int"/>
        <xsd:element name="varFloat" type="xsd:float"/>
       <xsd:element name="varStruct" type="tns:SimpleStruct"/>
    </xsd:sequence>
</xsd:complexType>
```

IDL mapping

Example 34 shows the result of mapping the `NestedStruct` type (from the preceding Example 33) to IDL.

Example 34: *Mapping of NestedStruct to IDL*

```
// IDL
struct SimpleStruct {
    float varFloat;
    long varInt;
    string varString;
};

struct NestedStruct {
    string varString;
    long varInt;
    float varFloat;
    SimpleStruct varStruct;
};
```

Deriving a Complex Type from a Simple Type

A complex type derived from a simple type maps to an IDL `struct` type with a member, `_simpleTypeValue`, to hold the value of the simple type. Any attributes defined by the derived type are represented as nillable members of the struct (see "Attributes" on page 101 for more details).

The following kinds of derivation are supported:

- Derivation by restriction.
- Derivation by extension.

Derivation by restriction

Example 35 shows an example of a complex type, `OrderNumber`, derived by restriction from the `xsd:decimal` simple type. The new type is restricted to have values less than 1,000,000.

Example 35: *Complex Type Derived by Restriction from a Simple Type*

```
<xsd:complexType name="OrderNumber">
    <xsd:simpleContent>
        <xsd:restriction base="xsd:decimal">
            <xsd:maxExclusive value="1000000"/>
        </xsd:restriction>
    </xsd:simpleContent>
</xsd:complexType>
```

IDL mapping of restricted type

Example 36 shows the result of mapping the `OrderNumber` type (from the preceding Example 35) to IDL. The `_simpleTypeValue` struct member represents the simple type value.

Example 36: *Mapping of OrderNumber to IDL*

```
// IDL
typedef fixed<31, 6> fixed_1;

struct OrderNumber {
    fixed_1 _simpleTypeValue;
};
```

Derivation by extension

Example 37 shows an example of a complex type, `InternationalPrice`, derived by extension from the `xsd:decimal` simple type. The new type is extended to include a currency attribute.

Example 37: *Complex Type Derived by Extension from a Simple Type*

```
<xsd:complexType name="InternationalPrice">
    <xsd:simpleContent>
        <xsd:extension base="xsd:decimal">
            <xsd:attribute name="currency"
   type="xsd:string"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
```

IDL mapping of extended type

Example 38 shows the result of mapping the `InternationalPrice` type (from the preceding Example 37) to IDL. In addition to the `_simpleTypeValue` member, representing the simple type, there is a `currency` member of `string_nil` type, representing the currency attribute.

Example 38: *Mapping of InternationalPrice to IDL*

```
// IDL
union string_nil switch(boolean) {
    case TRUE:
        string value;
};
typedef fixed<31, 6> fixed_1;

struct InternationalPrice {
    string_nil currency;
    fixed_1 _simpleTypeValue;
};
```

Deriving a Complex Type from a Complex Type

Artix supports derivation of a complex type from a complex type, for which the following kinds of derivation are possible:

- Derivation by restriction.
- Derivation by extension.

Allowed inheritance relationships

Figure 18 shows the inheritance relationships allowed between complex types. All of these inheritance relationships are supported by the WSDL-to-IDL mapping, including cross-inheritance. For example, a sequence can derive from a choice, a choice from an all, an all from a choice, and so on.

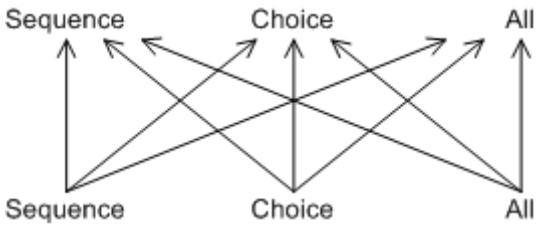

Figure 18: *Allowed Inheritance Relationships for Complex Types*

IDL mapping

Artix maps schema derived types to an IDL struct (irrespective of whether the schema derived type is a `sequence`, a `choice`, or an `all`). The generated IDL struct always contains the following two members:

- *The base member*—holds an instance of the base type, *BaseType*. The name of this member is *BaseType*`_f`.
- *The extension member*—holds an instance of the extension type. The name of this member obeys the following naming convention (where *DerivedType* is the name of the derived type in XML):
  - `sequence` extension—the name is *DerivedType*`SequenceStruct_f`.
  - `choice` extension—the name is *DerivedType*`ChoiceType_f`.
  - `all` extension—the name is *DerivedType*`AllStruct_f`.

In addition, if the derived type defines attributes, they are mapped directly to members of the IDL struct.

WSDL example

Example 39 shows the definition of a derived type that is obtained by extending a `sequence` type (base type) with a `choice` type (extension type).

Example 39: *XML Example of a Choice Type Derived from a Struct Type*

```
// Base type.
<xsd:complexType name="SimpleStruct">
    <xsd:sequence>
        <xsd:element name="varFloat" type="xsd:float"/>
        <xsd:element name="varInt" type="xsd:int"/>
        <xsd:element name="varString" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>

// Derived type.
<xsd:complexType name="DerivedChoice_BaseStruct">
    <xsd:complexContent mixed="false">
       <xsd:extension base="s:SimpleStruct">
            <xsd:choice>
                <xsd:element name="varStringExt"
                             type="xsd:string"/>
                <xsd:element name="varFloatExt"
   type="xsd:float"/>
            </xsd:choice>
           <xsd:attribute name="attrString" type="xsd:string"/>
       </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Mapped example

The preceding `DerivedChoice_BaseStruct` schema type maps to an IDL struct, `DerivedChoice_BaseStruct`, as shown in Example 40.

Example 40: *IDL Mapping of the DerivedChoice_BaseStruct Type*

```
// IDL

// Base type.
struct SimpleStruct {
    float varFloat;
    long varInt;
    string varString;
};

// Extended part of derived type.
union DerivedChoice_BaseStructChoiceType switch(long) {
    case 0:
        string varStringExt;
    case 1:
        float varFloatExt;
};

// Derived type.
struct DerivedChoice_BaseStruct {
    string_nil attrString;

    SimpleStruct SimpleStruct_f;
    DerivedChoice_BaseStructChoiceType
        DerivedChoice_BaseStructChoiceType_f;
};
```

Arrays

An Artix array is a sequence complex type that satisfies the following special conditions:

- The sequence complex type schema defines a *single* element only.
- The element definition has a `maxOccurs` attribute with a value greater than 1.

Note: All elements implicitly have `minOccurs=1` and `maxOccurs=1`, unless specified otherwise.

Hence, an Artix array definition has the following general syntax:

```
<complexType name="ArrayName">
    <sequence>
        <element name="ElemName" type="ElemType"
                 minOccurs="LowerBound" maxOccurs="UpperBound"/>
    </sequence>
</complexType>
```

The *ElemType* specifies the type of the array elements and the number of elements in the array can be anywhere in the range *LowerBound* to *UpperBound*.

Mapping arrays to IDL

The way Artix maps arrays to IDL depend on the values of the `minOccurs` and `maxOccurs` attributes, as shown in Table 3.

Table 3: *Array to IDL Mapping for Various Occurrence Constraints*

Occurrence Constraints	IDL Type
`minOccurs="`*N*`"` `maxOccurs="`*N*`"`	*ArrayName*[*N*]

`minOccurs="`*N*`"`
`maxOccurs="`*M*`"`
(with *N* < *M*)
	sequence<*ElementType*, *M*>

`maxOccurs="unbounded"`	sequence<*ElementType*>

Fixed array

The following XSD schema shows the definition of an array, `FixedArray`, whose `minOccurs` and `maxOccurs` constraints are set to an identical, finite value.

```
<xsd:complexType name="FixedArray">
    <xsd:sequence>
        <xsd:element maxOccurs="3" minOccurs="3"
                     name="item" type="xsd:int"/>
    </xsd:sequence>
</xsd:complexType>
```

The preceding `FixedArray` schema type maps to the following IDL array:

```
// IDL
typedef long FixedArray[3];
```

Bounded array

The following XSD schema shows the definition of an array, `BoundedArray`, whose `minOccurs` and `maxOccurs` constraints are finite and unequal.

```
<xsd:complexType name="BoundedArray">
    <xsd:sequence>
        <xsd:element maxOccurs="3" minOccurs="1"
                     name="item" type="xsd:float"/>
    </xsd:sequence>
</xsd:complexType>
```

The preceding `BoundedArray` schema type maps to the following IDL bounded sequence type:

```
// IDL
typedef sequence<float, 3> BoundedArray;
```

Unbounded array

The following XSD schema shows the definition of an array, `UnboundedArray`, whose `maxOccurs` constraint is unbounded.

```
<xsd:complexType name="UnboundedArray">
    <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0"
                     name="item" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

The preceding `UnboundedArray` schema type maps to the following IDL unbounded sequence type:

```
// IDL
typedef sequence<string> UnboundedArray;
```

Nested arrays

The following XSD schema shows the definition of a nested array, `NestedArray`, which is defined as an array whose elements are of `UnboundedArray` type.

```
<xsd:complexType name="NestedArray">
    <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0"
                     name="subarray"
   type="s:UnboundedArray"/>
    </xsd:sequence>
</xsd:complexType>
```

The preceding `NestedArray` schema type maps to the following IDL unbounded sequence type:

```
// IDL
typedef sequence<UnboundedArray> NestedArray;
```

Wildcarding Types

The XML schema wildcarding types enable you to define XML types with loosely defined characteristics. Table 4 shows how the XSD schema wildcarding types map to IDL.

Table 4: *XSD Schema Simple Types Mapping to IDL*

XSD Schema Type	IDL Type
`xsd:anyURI`	`string`
`xsd:anyType`	`any`
`xsd:any`	*Not supported*

xsd:anyType example

Consider an XSD sequence, `AnyStruct`, whose elements are declared to be of `xsd:anyType` type, as shown in Example 41.

Example 41: *AnyStruct Schema Type with xsd:anyType Members*

```
<xsd:complexType name="AnyStruct">
    <xsd:sequence>
        <xsd:element name="varAny_1" type="xsd:anyType"/>
        <xsd:element name="varAny_2" type="xsd:anyType"/>
    </xsd:sequence>
</xsd:complexType>
```

The preceding `AnyStruct` schema type maps to the IDL struct type shown in Example 42.

Example 42: *Mapping of AnyStruct Type to IDL*

```
struct AnyStruct {
    any varAny_1;
    any varAny_2;
};
```

Occurrence Constraints

Certain XML schema tags—for example, `<element>`, `<sequence>`, `<choice>` and `<any>`—can be declared to occur multiple times using *occurrence constraints*. The occurrence constraints are specified by assigning integer values (or the special value `unbounded`) to the `minOccurs` and `maxOccurs` attributes.

The WSDL-to-IDL mapping currently supports only *element occurrence constraints* (that is, `minOccurs` and `maxOccurs` attribute settings within the `<element>` tag).

Element occurrence constraints

You define occurrence constraints on a schema element by setting the `minOccurs` and `maxOccurs` attributes for the element. Hence, the definition of an element with occurrence constraints in an XML schema element has the following form:

```
<element name="ElemName" type="ElemType" minOccurs="LowerBound"
   maxOccurs="UpperBound"/>
```

Note: When a sequence schema contains a *single* element definition and this element defines occurrence constraints, it is treated as an array. See "Arrays" on page 107.

Limitations

In the current version of Artix, element occurrence constraints can be used only within the following complex types:

- `all` complex types,
- `sequence` complex types.

Element occurrence constraints are *not* supported within the scope of the following:

- `choice` complex types.

Mapping to IDL

Given an `<xsd:element name="`*ElemName*`" ... >` element with occurrence constraints, defined in an `<xsd:sequence>` or an `<xsd:all>` tag, Artix defines an *ElemName*`Array` type in IDL to represent the multiply occurring element.

The *ElemName*`Array` type is defined according to the rules in Table 3 on page 108, which determine the mapped IDL type based on the values of the `minOccurs` and `maxOccurs` attributes.

Example of element occurrence constraints

The following XSD schema shows the definition of an `<xsd:sequence>` type, `CompoundArray`, which has two multiply occurring member elements.

```
<xsd:complexType name="CompoundArray">
    <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0"
                     name="array1" type="xsd:string"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0"
                     name="array2" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

The preceding `CompoundArray` schema type maps to the following IDL struct, `CompoundArray`, which uses two generated array types, `array1Array` and `array2Array`, to represent the types of its member elements:

```
// IDL
typedef sequence<string> array1Array;
typedef sequence<string> array2Array;

struct CompoundArray {
    array1Array array1;
    array2Array array2;
};
```

Nillable Types

An element in an XML schema may be declared as nillable by setting the `nillable` attribute equal to `true`. This is useful in cases where you would like to have the option of transmitting no value for a type (for example, if you would like to define an operation with optional parameters).

Nillable syntax

To declare an element as nillable, use the following syntax:

`<element name="`*ElementName*`" type="`*ElementType*`" nillable="true"/>`

The `nillable="true"` setting indicates that this as a nillable element. If the `nillable` attribute is missing, the default is value is `false`.

Mapping to IDL

If a given element of *ElementType* type is defined with `nillable="true"` and *ElementType* maps to *MappedType* in IDL, Artix automatically generates a union IDL type, *MappedType*`_nil`, as follows:

```
// IDL
union MappedType_nil switch(boolean) {
    case TRUE:
        MappedType value;
};
```

Artix uses this *MappedType*`_nil` type to represent the type of the nillable element in IDL (for example, where it appears as the member of a struct and so on).

Example

The following XSD schema shows the definition of an `<xsd:sequence>` type, `StructWithNillables`, which contains several nillable elements:

```
<xsd:complexType name="SimpleStruct">
    <xsd:sequence>
        <xsd:element name="varFloat" type="xsd:float"/>
        <xsd:element name="varInt" type="xsd:int"/>
        <xsd:element name="varString" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="varAttrString" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="StructWithNillables">
    <xsd:sequence>
        <xsd:element name="varFloat" nillable="true"
                     type="xsd:float"/>
        <xsd:element name="varInt" nillable="true"
                     type="xsd:int"/>
```

```
        <xsd:element name="varString" nillable="true"
                     type="xsd:string"/>
        <xsd:element name="varStruct" nillable="true"
                     type="s:SimpleStruct"/>
    </xsd:sequence>
</xsd:complexType>
```

The preceding `StructWithNillables` schema type maps to the IDL struct, `StructWithNillables`, which uses generated nillable types, `float_nil`, `long_nil`, `string_nil` and `SimpleStruct_nil`, to represent the types of its member elements:

```
// IDL
union float_nil switch(boolean) {
    case TRUE:
        float value;
};
union long_nil switch(boolean) {
    case TRUE:
        long value;
};
union string_nil switch(boolean) {
    case TRUE:
        string value;
};

struct SimpleStruct {
    string_nil varAttrString;
    float varFloat;
    long varInt;
    string varString;
};
union SimpleStruct_nil switch(boolean) {
    case TRUE:
        SimpleStruct value;
};

struct StructWithNillables {
    float_nil varFloat;
    long_nil varInt;
    string_nil varString;
    SimpleStruct_nil varStruct;
};
```

Recursive Types

XML schema allows you to define *recursive* types and the WSDL-to-IDL compiler is able to map these types into OMG IDL. The following kinds of recursive type are considered here:

- *Self-recursive types*—a type that refers to itself within its own definition.
- *Mutually-recursive types*—for example, given two types, A and B, the definition of A refers to B and the definition of B refers to A.

  More complex recursions are also supported—for example, where A refers to B refers to C refers to A (in shorthand, `A ->`

B -> C -> A). Overlapping recursions are also supported—for example, A -> C -> A and A -> B -> C -> A at the same time.

Note: Mutual recursion does not work, however, in cases where the recursive types are defined in separate IDL modules. See "Circular references across modules" on page 127.

The IDL mapping of recursive types relies on the use of forward declarations of IDL structs.

Note: Forward declaration of structs is a relatively new feature of the OMG IDL syntax and might not be supported by all ORB products.

Complex types that can use recursion

The following complex XML schema types can be defined with recursion:

- xsd:sequence,
- xsd:union,
- xsd:all.

XML schema example of self-recursive type

Example 43 shows an example of a self-recursive sequence—that is, a sequence type, RecurSeq, that contains a reference to itself.

Example 43: *XML Example of a Self-Recursive Type*

```
<xsd:complexType name="RecurSeq">
    <xsd:sequence>
        <xsd:element name="value" type="xsd:long"/>
        <xsd:element name="RecurSeqs" type="s:RecurSeq"
                     minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
```

Note: In this example, it is important to set minOccurs equal to zero, otherwise the recursion could never terminate.

IDL mapping of self-recursive type

Example 44 shows how the self-recursive type, `RecurSeq`, (from Example 43 on page 114) maps to OMG IDL. This mapping uses a forward declaration of the `RecurSeq` IDL struct to define the recursive type.

Example 44: *IDL Mapping of a Self-Recursive Type*

```
// IDL
struct RecurSeq;
typedef sequence<RecurSeq> RecurSeqsArray;

struct RecurSeq {
    long long value;
    RecurSeqsArray RecurSeqs;
};
```

Note: Forward declaration of an OMG IDL struct is supported only by Orbix version and later.

XML schema example of mutually-recursive types

Example 45 shows an example of two mutually-recursive sequence types, `MutualSeqA` and `MutualSeqB`. In this example, `MutualSeqB` contains a reference to `MutualSeqA` and `MutualSeqA` contains a reference to `MutualSeqB`.

Example 45: *XML Example of Mutually-Recursive Types*

```
<xsd:complexType name="MutualSeqA">
    <xsd:sequence>
        <xsd:element name="valueA" type="xsd:long"/>
       <xsd:element name="MutualSeqBs" type="s:MutualSeqB"
                    minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="MutualSeqB">
    <xsd:sequence>
        <xsd:element name="OneMutualSeqA"
   type="s:MutualSeqA"/>
        <xsd:element name="valueB" type="xsd:long"/>
    </xsd:sequence>
</xsd:complexType>
```

IDL mapping of mutually-recursive types

Example 46 shows how the mutually-recursive types, `MutualSeqA` and `MutualSeqB` (from Example 45 on page 115) map to OMG IDL. This mapping uses forward declarations of the `MutualSeqA` struct and the `MutualSeqB` struct, in order to define the recursive types.

Example 46: *IDL Mapping of Mutually-Recursive Types*

```
// IDL
struct MutualSeqB;
struct MutualSeqA;
typedef sequence<MutualSeqB> MutualSeqBsArray;

struct MutualSeqA {
    long long valueA;
    MutualSeqBsArray MutualSeqBs;
};
struct MutualSeqB {
    MutualSeqA OneMutualSeqA;
    long long valueB;
};
```

Endpoint References

Endpoint references provide a convenient way of encapsulating the location of an Artix service, in a form that can be passed as a parameter or a return value in a WSDL operation. In the special case *where the endpoint reference refers to a CORBA port*, it is possible to map the endpoint reference to a CORBA object reference. It is obviously *not* possible for a CORBA client to use an object reference to connect to a non-CORBA service.

Endpoint reference type

The endpoint reference type is defined by the WS-Addressing standard. In Artix, the endpoint reference type is normally represented as `wsa:EndpointReferenceType`.

WS-Addressing namespace

Artix conventionally defines the namespace prefix, `wsa`, to represent the WS-Addressing namespace:

`http://www.w3.org/2005/08/addressing`

To use endpoint references, you should define the `wsa` namespace prefix in the `definitions` element of your WSDL contract.

WS-Addressing schema import

In order to use endpoint references in a WSDL contract, you must also import the WS-Addressing schema, using the following import statement:

```
<import namespace="http://www.w3.org/2005/08/addressing"
        schemaLocation="WSAddressingURL"/>
```

Where *WSAddressingURL* can either be the path to an `.xsd` file in the local filesystem or a URL to retrieve the schema from a remote location.

Default Endpoint Reference Mapping

By default, the endpoint reference type, `wsa:EndpointReferenceType`, maps to the IDL built-in type, `Object`.

Using an endpoint reference type

To use an endpoint reference in your contract, simply declare a parameter or return value to be of `wsa:EndpointReferenceType` in an operation's request or reply message. For example, to declare the return value from a `create_account` operation to be an endpoint reference type, you would define the operation's request and reply messages as follows:

Example 47: *Request and Reply Messages for create_account Operation*

```
<message name="create_account">
    <part name="account_name" type="xsd:string"/>
</message>
<message name="create_accountResponse">
    <part name="return" type="wsa:EndpointReferenceType"/>
</message>
```

Empty endpoint reference

An *empty endpoint reference* is an endpoint reference that does not address any endpoint. Conventionally, the address of an empty endpoint reference is represented by the following URI:

```
http://www.w3.org/2005/08/addressing/none
```

Artix maps an empty endpoint reference to a CORBA nil object reference.

WSDL example

Example 48 shows how endpoint references are used in a bank WSDL contract. The Bank service exposes two operations, `create_account` and `get_account`, which return references to

Account services. The returned references are declared to be of endpoint reference type, `wsa:EndpointReferenceType` (highlighted in bold font).

Example 48: *Example Using Default Mapping of EndpointReferenceType*

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://www.iona.com/bus/demos/bank"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:xsd1="http://soapinterop.org/xsd"
             xmlns:stub="http://schemas.iona.com/transports/stub"
             xmlns:http="http://schemas.iona.com/transports/http"
             xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"
             xmlns:fixed="http://schemas.iona.com/bindings/fixed"
             xmlns:iiop="http://schemas.iona.com/transports/iiop_tunnel"
             xmlns:corba="http://schemas.iona.com/bindings/corba"
             xmlns:ns1="http://www.iona.com/corba/typemap/BasePortType.idl"
            xmlns:wsa="http://www.w3.org/2005/08/addressing"
             xmlns:mq="http://schemas.iona.com/transports/mq"
             xmlns:routing="http://schemas.iona.com/routing"
             xmlns:msg="http://schemas.iona.com/port/messaging"
             xmlns:bank="http://www.iona.com/bus/demos/bank"
             targetNamespace="http://www.iona.com/bus/demos/bank"
             name="BaseService" >
    <types>

        <schema elementFormDefault="qualified"
                targetNamespace="http://www.iona.com/bus/demos/bank"
                xmlns="http://www.w3.org/2001/XMLSchema">

           <import namespace="http://www.w3.org/2005/08/addressing"
                   schemaLocation="wsaddressing.xsd"/>

            <complexType name="AccountNames">
                <sequence>
                    <element maxOccurs="unbounded" minOccurs="0" name="name"
   type="xsd:string"/>
                </sequence>
            </complexType>
        </schema>
    </types>

    <message name="list_accounts" />
    <message name="list_accountsResponse">
        <part name="return" type="bank:AccountNames"/>
    </message>

    <message name="create_account">
        <part name="account_name" type="xsd:string"/>
    </message>
    <message name="create_accountResponse">
        <part name="return" type="wsa:EndpointReferenceType"/>
    </message>

    <message name="get_account">
        <part name="account_name" type="xsd:string"/>
    </message>
    <message name="get_accountResponse">
```

Example 48: *Example Using Default Mapping of EndpointReferenceType*

```
        <part name="return" type="wsa:EndpointReferenceType"/>
    </message>

    <message name="delete_account">
        <part name="account_name" type="xsd:string"/>
    </message>
    <message name="delete_accountResponse" />

    <message name="get_balance"/>
    <message name="get_balanceResponse">
        <part name="balance" type="xsd:float"/>
    </message>

    <message name="deposit">
        <part name="addition" type="xsd:float"/>
    </message>

    <message name="depositResponse"/>


    <portType name="Bank">

        <operation name="list_accounts">
            <input name="list_accounts" message="tns:create_account"/>
            <output name="list_accountsResponse" message="tns:list_accountsResponse"/>
        </operation>

        <operation name="create_account">
            <input name="create_account" message="tns:create_account"/>
            <output name="create_accountResponse"
   message="tns:create_accountResponse"/>
        </operation>

        <operation name="get_account">
            <input name="get_account" message="tns:get_account"/>
            <output name="get_accountResponse" message="tns:get_accountResponse"/>
        </operation>

        <operation name="delete_account">
            <input name="delete_account" message="tns:delete_account"/>
            <output name="delete_accountResponse"
   message="tns:delete_accountResponse"/>
        </operation>

    </portType>
    <portType name="Account">
        <operation name="get_balance">
            <input name="get_balance" message="tns:get_balance"/>
            <output name="get_balanceResponse" message="tns:get_balanceResponse"/>
        </operation>
        <operation name="deposit">
            <input name="deposit" message="tns:deposit"/>
            <output name="depositResponse" message="tns:depositResponse"/>
        </operation>
    </portType>
    ...
</definitions>
```

IDL mapping

When the preceding WSDL contract (Example 48 on page 118) is mapped to OMG IDL, the Bank operations are mapped as shown in Example 49.

Example 49: *Bank Interface with Default Endpoint Reference Mapping*

```
// IDL
interface Bank {
    ::AccountNames
    list_accounts(
        in string account_name
    );
    Object
    create_account(
        in string account_name
    );
    Object
    get_account(
        in string account_name
    );
    void
    delete_account(
        in string account_name
    );
};
```

Custom Endpoint Reference Mapping

Whereas in WSDL all endpoint references must be of the same type (that is, `wsa:EndpointReferenceType`), in IDL object references are usually declared as type specific. For example, if an IDL operation returns a reference to an account, the return value is normally defined to be of `Account` type, rather than the generic `Object` type.

To ensure that a WSDL endpoint reference maps to a type-specific object reference in IDL, you can add an annotation to the WSDL contract.

Annotation for a custom endpoint reference mapping

To customize the mapping of an endpoint reference, you must modify the parameters in an operation's request or reply message to refer to a custom element instead of referring to a `wsa:EndpointReferenceType` type. The custom element must then be defined with an `xsd:annotation` element that contains details of the custom mapping.

For example, Example 50 shows you how to define a reply message for the `create_account` operation from the Bank WSDL contract, such that the type returned from `create_account` maps to `Account` in IDL.

Example 50: *Annotation for Custom Mapping of Endpoint Reference*

```
<types>
    <schema ... >
        ...
       <element name="AccountRef"
                type="wsa:EndpointReferenceType">
          <annotation>
           <appinfo>corba:binding=AccountCORBABinding</appinfo>
          </annotation>
       </element>
    </schema>
</types>

<message name="create_account">
    <part name="account_name" type="xsd:string"/>
</message>
<message name="create_accountResponse">
    <part element="bank:AccountRef" name="return"/>
</message>
```

The annotation in the `AccountRef` element is defined in order to map the `wsa:EndpointReferenceType` to the `Account` interface. The setting in the `<appinfo>` tag:

```
corba:binding=BindingName
```

identifies an associated `Account` binding, rather than an `Account` port type, because the annotation applies specifically to the CORBA binding, not to all bindings.

WSDL example

Example 51 shows an example of a Bank WSDL contract that uses an annotation to customize the mapping of the endpoint reference type.

Example 51: *Example Using Custom Mapping of EndpointReferenceType*

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://www.iona.com/bus/demos/bank"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:xsd1="http://soapinterop.org/xsd"
             xmlns:stub="http://schemas.iona.com/transports/stub"
             xmlns:http="http://schemas.iona.com/transports/http"
             xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"
             xmlns:fixed="http://schemas.iona.com/bindings/fixed"
             xmlns:iiop="http://schemas.iona.com/transports/iiop_tunnel"
             xmlns:corba="http://schemas.iona.com/bindings/corba"
             xmlns:ns1="http://www.iona.com/corba/typemap/BasePortType.idl"
            xmlns:wsa="http://www.w3.org/2005/08/addressing"
```

Example 51: *Example Using Custom Mapping of EndpointReferenceType*

```
            xmlns:mq="http://schemas.iona.com/transports/mq"
            xmlns:routing="http://schemas.iona.com/routing"
            xmlns:msg="http://schemas.iona.com/port/messaging"
            xmlns:bank="http://www.iona.com/bus/demos/bank"
            targetNamespace="http://www.iona.com/bus/demos/bank"
            name="BaseService" >
  <types>

      <schema elementFormDefault="qualified"
              targetNamespace="http://www.iona.com/bus/demos/bank"
              xmlns="http://www.w3.org/2001/XMLSchema">

         <import namespace="http://www.w3.org/2005/08/addressing"
                 schemaLocation="wsaddressing.xsd"/>

           <complexType name="AccountNames">
               <sequence>
                   <element maxOccurs="unbounded" minOccurs="0" name="name"
type="xsd:string"/>
               </sequence>
           </complexType>

         <xsd:element name="AccountRef" type="wsa:EndpointReferenceType">
             <xsd:annotation>
                 <xsd:appinfo>corba:binding=AccountCORBABinding</xsd:appinfo>
             </xsd:annotation>
         </xsd:element>
      </schema>
  </types>

  <message name="list_accounts" />
  <message name="list_accountsResponse">
      <part name="return" type="bank:AccountNames"/>
  </message>

  <message name="create_account">
      <part name="account_name" type="xsd:string"/>
  </message>
  <message name="create_accountResponse">
      <part name="return" element="tns:AccountRef"/>
  </message>

  <message name="get_account">
      <part name="account_name" type="xsd:string"/>
  </message>
  <message name="get_accountResponse">
      <part name="return" element="tns:AccountRef"/>
  </message>

  <message name="delete_account">
      <part name="account_name" type="xsd:string"/>
  </message>
  <message name="delete_accountResponse" />

  <message name="get_balance"/>
  <message name="get_balanceResponse">
      <part name="balance" type="xsd:float"/>
  </message>
```

Example 51: *Example Using Custom Mapping of EndpointReferenceType*

```
    <message name="deposit">
        <part name="addition" type="xsd:float"/>
    </message>

    <message name="depositResponse"/>


    <portType name="Bank">

        <operation name="list_accounts">
            <input name="list_accounts" message="tns:create_account"/>
            <output name="list_accountsResponse" message="tns:list_accountsResponse"/>
        </operation>

        <operation name="create_account">
            <input name="create_account" message="tns:create_account"/>
            <output name="create_accountResponse"
   message="tns:create_accountResponse"/>
        </operation>

        <operation name="get_account">
            <input name="get_account" message="tns:get_account"/>
            <output name="get_accountResponse" message="tns:get_accountResponse"/>
        </operation>

        <operation name="delete_account">
            <input name="delete_account" message="tns:delete_account"/>
            <output name="delete_accountResponse"
   message="tns:delete_accountResponse"/>
        </operation>

    </portType>
    <portType name="Account">
        <operation name="get_balance">
            <input name="get_balance" message="tns:get_balance"/>
            <output name="get_balanceResponse" message="tns:get_balanceResponse"/>
        </operation>
        <operation name="deposit">
            <input name="deposit" message="tns:deposit"/>
            <output name="depositResponse" message="tns:depositResponse"/>
        </operation>
    </portType>
    ...
</definitions>
```

Generating the IDL interfaces

To generate IDL from the WSDL contract shown in Example 51 on page 121, perform the following steps:

1. Generate the CORBA binding for the Account interface, using the following command:

   C++ runtime

   ```
   wsdltocorba -corba -i Account -b AccountCORBABinding
      bank.wsdl
   ```

   Where the bank WSDL contract is stored in the file, `bank.wsdl`. The output from this command is a new WSDL file, `bank-corba.wsdl`, which includes the `AccountCORBABinding` binding.
2. Generate the CORBA binding for the Bank interface, using the following command:

   C++ runtime

   ```
   wsdltocorba -corba -i Bank -b BankCORBABinding
      -o bank-corba2.wsdl bank-corba.wsdl
   ```

   The output from this command is a new WSDL file, `bank-corba2.wsdl`, which includes both the `AccountCORBABinding` binding and the `BankCORBABinding` binding.

   Note: The order in which these two commands are issued is important, because the `BankCORBABinding` binding references the `AccountCORBABinding` binding.

3. Convert the WSDL contract with CORBA bindings into IDL, using the following command:

   C++ runtime

   ```
   wsdltocorba -idl -b BankCORBABinding bank-corba2.wsdl
   ```

CORBA type mapping

Example 52 shows the generated CORBA type mapping that results from adding both the `AccountCORBABinding` and the `BankCORBABinding` into the contract.

Example 52: *CORBA Type Mapping with References*

```
<corba:typeMapping
    targetNamespace="http://www.iona.com/bus/demos/bank/corba/typemap/">
...
    <corba:object binding="" name="Object"
                  repositoryID="IDL:omg.org/CORBA/Object/1.0"
   type="wsa:EndpointReferenceType"/>
    <corba:object binding="AccountCORBABinding" name="Account"
                  repositoryID="IDL:Account:1.0" type="wsa:EndpointReferenceType"/>
</corba:typeMapping>
```

There are two entries because `wsdltocorba` was run twice on the same file. The first CORBA object is generated from the first pass of `wsdltocorba` to generate the CORBA binding for `Account`. Because `wsdltocorba` could not find the binding specified in the annotation, it generated a generic `Object` reference. The second CORBA object, `Account`, is generated by the second pass when the binding for `Bank` was generated. On that pass, `wsldtocorba` could inspect the binding for the `Account` interface and generate a type-specific object reference.

IDL mapping

Example 53 shows the IDL generated for the `Account` and `Bank` interfaces.

Example 53: *IDL Generated From Artix References*

```
//IDL
...
interface Account {
    float
    get_balance();
    void
    deposit(
        in float addition
    );
};
interface Bank {
    ::AccountNames
    list_accounts(
        in string account_name
    );
    ::Account
    create_account(
        in string account_name
    );
    ::Account
    get_account(
        in string account_name
    );
    void
    delete_account(
        in string account_name
    );
};
```

Mapping to IDL Modules

If you want your generated IDL files to be organized into modules, you can achieve this by applying the appropriate naming convention to the XML schema types that appear in the WSDL contract. The following aspects of IDL module mapping are discussed in this section:

- Module mapping convention.
- References across modules.
- Circular references across modules.

Module mapping convention

In order to indicate to the Artix WSDL-to-IDL compiler that you want a type to appear inside an IDL module, give the type a local name with the following compound format:

ModuleName_1.*ModuleName_2*. ... .*ModuleName_N*.*TypeName*

Where Artix uses the period character, ., as a delimiter. *ModuleName_1* to *ModuleName_N* are the names of a series of nested IDL modules and *TypeName* is the unscoped type name in IDL.

For example, you can define an XML sequence type with the compound name, `ONE.SeqA`, as follows:

```
<xsd:complexType name="ONE.SeqA">
    <xsd:sequence>
        <xsd:element name="valueA" type="xsd:long"/>
    </xsd:sequence>
</xsd:complexType>
```

When you map this data type to IDL, you obtain a module, `ONE`, containing a struct definition, `SeqA`, as follows:

```
// IDL
module ONE {
    struct SeqA {
        long long valueA;
    };
};
```

References across modules

It is also possible to make references across modules. That is, a type defined in one module can use the elements or types defined in another module.

For example, you can define an XML sequence, `ONE.SeqA`, which has a member whose type is that of another sequence, `TWO.SeqB`, as follows:

```
<xsd:complexType name="ONE.SeqA">
    <xsd:sequence>
        <xsd:element name="valueA" type="xsd:long"/>
        <xsd:element name="seqB" type="s:TWO.SeqB"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="TWO.SeqB">
    <xsd:sequence>
        <xsd:element name="valueB" type="xsd:long"/>
    </xsd:sequence>
</xsd:complexType>
```

When you map the preceding types to IDL, the `seqB` member of the `SeqA` struct is of a type, `::TWO::SeqB`, that is defined in the second module, as follows:

```
// IDL
module TWO {
    struct SeqB {
        long long valueB;
    };
};
module ONE {
    struct SeqA {
        long long valueA;
        ::TWO::SeqB seqB;
    };
};
```

Circular references across modules

Artix currently does *not* support the case where you have a chain of references between modules that form a closed loop.

For example, the following XML schema fragment—where the `ONE.SeqA` sequence references the `TWO.SeqB` sequence, which references the `ONE.SeqC` sequence—is not supported:

```
<xsd:complexType name="ONE.SeqA">
    <xsd:sequence>
        <xsd:element name="valueA" type="xsd:long"/>
        <xsd:element name="seqB" type="s:TWO.SeqB"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="TWO.SeqB">
    <xsd:sequence>
        <xsd:element name="seqC" type="s:ONE.SeqC"/>
        <xsd:element name="valueB" type="xsd:long"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ONE.SeqC">
    <xsd:sequence>
        <xsd:element name="valueC" type="xsd:long"/>
    </xsd:sequence>
</xsd:complexType>
```

If you map the preceding WSDL example to IDL, and then map the IDL to C++, you obtain stub code that is *not* compilable (the IDL is missing a forward reference to a struct).

Monitoring GIOP Message Content

Artix includes the GIOP Snoop tool for intercepting and displaying GIOP message content.

WARNING: It is recommended that you avoid using this feature in secure applications. The GIOP snoop plug-in can expose user names and passwords.

Introduction to GIOP Snoop

GIOP Snoop is a GIOP protocol level plug-in for intercepting and displaying GIOP message content. This plug-in implements message level interceptors that can participate in client and/or server side bindings over any GIOP-based transport. The primary purposes of GIOP Snoop are to provide a protocol level monitor and debug aid.

WARNING: It is recommended that you avoid using this feature in secure applications. The GIOP snoop plug-in can expose user names and passwords.

GIOP plug-ins

The primary protocol for inter-ORB communications is the General Inter-ORB Protocol (GIOP) as defined the CORBA Specification.

Configuring GIOP Snoop

GIOP Snoop can be configured for debugging in client, server, or both depending on configuration. This section includes the following configuration topics:

- Loading the GIOP Snoop plug-in.
- Client-side snooping.
- Server-side snooping.
- GIOP Snoop verbosity levels.
- Directing output to a file.

Loading the GIOP Snoop plug-in

For either client or server configuration, the GIOP Snoop plug-in must be included in the Orbix `orb_plugins` list (. . . denotes existing configured settings):

```
orb_plugins = [..., "giop_snoop", ...];
```

In addition, the giop_snoop plug-in must be located and loaded using the following settings:

```
# Artix Configuration File
plugins:giop_snoop:shlib_name = "it_giop_snoop";
```

Client-side snooping

To enable client-side snooping, include the GIOP_SNOOP factory in the client binding list. In this example, GIOP Snoop is enabled for IIOP-specific bindings:

```
binding:client_binding_list =
      [..., "GIOP+GIOP_SNOOP+IIOP", ...];
```

Server-side snooping

To enable server-side snooping, include the GIOP_SNOOP factory in the server binding list.

```
plugins:giop:message_server_binding_list =
     [..., "GIOP_SNOOP+GIOP", ...];
```

GIOP Snoop verbosity levels

You can use the following variable to control the GIOP Snoop verbosity level:

```
plugins:giop_snoop:verbosity = "1";
```

The verbosity levels are as follows:

```
1        LOW
2        MEDIUM
3        HIGH
4        VERY HIGH
```

These verbosity levels are explained with examples in “GIOP Snoop Output” on page 131.

Directing output to a file

By default, output is directed to standard error (`stderr`). However, you can specify an output file using the following configuration variable:

```
plugins:giop_snoop:filename = "<some-file-path>";
```

A month/day/year time stamp is included in the output filename with the following general format:

```
<filename>.MMDDYYYY
```

As a result, for a long running application, each day results in the creation of a new log file. To enable administrators to control the size and content of output files GIOP Snoop does not hold output files open. Instead, it opens and then closes the file for each snoop message trace. This setting is enabled with:

```
plugins:giop_snoop:rolling_file = "true";
```

WARNING: It is recommended that you avoid logging GIOP messages in secure applications. The GIOP snoop plug-in can expose user names and passwords.

GIOP Snoop Output

The output shown in this section uses a simple example that shows client-side output for a single binding and operation invocation. The client establishes a client-side binding that involves a message interceptor chain consisting of IIOP, GIOP Snoop, and GIOP. The client then connects to the server and first sends a `[LocateRequest]` to the server to test if the target object is reachable. When confirmed, a two-way invocation `[Request]` is sent, and the server processes the request. When complete, the server sends a `[Reply]` message back to the client.

Output detail varies depending on the configured verbosity level. With level `1` (`LOW`), only basic message type, direction, operation name and some GIOP header information (version, and so on) is given. More detailed output is possible, as described under the following examples.

LOW verbosity client-side snooping

An example of `LOW` verbosity output is as follows:

```
[Conn:1] Out:(first for binding)  [LocateRequest] MsgLen: 39
   ReqId: 0
[Conn:1] In: (first for binding)  [LocateReply] MsgLen: 8
   ReqId: 0
    Locate status: OBJECT_HERE
[Conn:1] Out:  [Request] MsgLen: 60 ReqId: 1  (two-way)
    Operation (len 8) 'null_op'
[Conn:1] In:   [Reply] MsgLen: 12 ReqId: 1
    Reply status (0) NO_EXCEPTION
```

This example shows an initial conversation from the client-side perspective. The client transmits a [LocateRequest] message to which it receives a [LocateReply] indicates that the server supports the target object. It then makes an invocation on the operation null_op.

The Conn indicates the logical connection. Because GIOP may be mapped to multiple transports, there is no transport specific information visible to interceptors above the transport (such as file descriptors) so each connection is given a logical identifier. The first incoming and outgoing GIOP message to pass through each connection are indicated by (first for binding).

The direction of the message is given (Out for outgoing, In for incoming), followed by the GIOP and message header contents. Specific information includes the GIOP version (version 1.2 above), message length and a unique request identifier (ReqId), which associates [LocateRequest] messages with their corresponding [LocateReply] messages. The (two-way) indicates the operation is two way and a response (Reply) is expected. String lengths such as len 8 specified for Operation includes the trailing null.

MEDIUM verbosity client-side snooping

An example of MEDIUM verbosity output is as follows:

```
16:24:39 [Conn:1] Out:(first for binding)  [LocateRequest]   GIOP v1.2  MsgLen: 39
   Endian: big  ReqId: 0
   Target Address (0: KeyAddr)
   ObjKey (len 27) ':>.11........\..A..........'

16:24:39 [Conn:1] In: (first for binding)  [LocateReply]     GIOP v1.2  MsgLen: 8
   Endian: big  ReqId: 0
   Locate status: OBJECT_HERE

16:24:39 [Conn:1] Out:  [Request]          GIOP v1.2  MsgLen: 60
   Endian: big  ReqId: 1  (two-way)
   Target Address (0: KeyAddr)
   ObjKey (len 27) ':>.11........\..A..........'
   Operation (len 8) 'null_op'

16:24:39 [Conn:1] In:   [Reply]            GIOP v1.2  MsgLen: 12
   Endian: big  ReqId: 1
   Reply status (0) NO_EXCEPTION
```

For MEDIUM verbosity output, extra information is provided. The addition of time stamps (in *hh:mm:ss*) precedes each snoop line. The byte order of the data is indicated (Endian) along with more detailed header information such as the target address shown in this example. The target address is a GIOP 1.2 addition in place of the previous object key data.

HIGH verbosity client side snooping

The following is an example of HIGH verbosity output:

```
16:24:39 [Conn:1] Out:(first for binding)  [LocateRequest]    GIOP v1.2  MsgLen: 39
    Endian: big  ReqId: 0
    Target Address (0: KeyAddr)
      ObjKey (len 27) ':>.11...........A.........'
    GIOP Hdr (len 12): [47][49][4f][50][01][02][00][03][00][00][00][27]
    Msg Hdr  (len 39): [00][00][00][00][00][00][00][00][00][00][00][1b][3a][3e]
[02][31][31][0c][00][00][00][00][00][00][0f][05][00][00][41][c6][08][00][00][00]
[00][00][00][00][00]
[---- end of message ----]

16:31:37 [Conn:1] In: (first for binding)  [LocateReply]      GIOP v1.2  MsgLen: 8
    Endian: big  ReqId: 0
    Locate status: OBJECT_HERE
    GIOP Hdr (len 12): [47][49][4f][50][01][02][00][04][00][00][00][08]
    Msg Hdr  (len 8): [00][00][00][00][00][00][00][01]
[---- end of message ----]

16:31:37 [Conn:1] Out:  [Request]         GIOP v1.2  MsgLen: 60
    Endian: big  ReqId: 1  (two-way)
    Target Address (0: KeyAddr)
      ObjKey (len 27) ':>.11...........A.........'
    Operation (len 8) 'null_op'
    No. of Service Contexts: 0
    GIOP Hdr (len 12): [47][49][4f][50][01][02][00][00][00][00][00][3c]
    Msg Hdr  (len 60): [00][00][00][01][03][00][00][00][00][00][00][00][00][00]
[00][1b][3a][3e][02][31][31][0c][00][00][00][00][00][00][0f][05][00][00][41][c6]
[08][00][00][00][00][00][00][00][00][00][00][00][00][08][6e][75][6c][6c][5f][6f]
[70][00][00][00][00][00]
[---- end of message ----]

16:31:37 [Conn:1] In:   [Reply]           GIOP v1.2  MsgLen: 12
    Endian: big  ReqId: 1
    Reply status (0) NO_EXCEPTION
    No. of Service Contexts: 0
    GIOP Hdr (len 12): [47][49][4f][50][01][02][00][01][00][00][00][0c]
    Msg Hdr  (len 12): [00][00][00][01][00][00][00][00][00][00][00][00]
[---- end of message ----]
```

This level of verbosity includes all header data, such as service context data. ASCII-hex pairs of GIOP header and message header content are given to show the exact on-the-wire header values passing through the interceptor. Messages are also separated showing inter-message boundaries.

VERY HIGH verbosity client side snooping

This is the highest verbosity level available. Displayed data includes HIGH level output and in addition the message body content is displayed. Because the plug-in does not have access to IDL interface definitions, it does not know the data types contained in the body (parameter values, return values and so on) and simply provides ASCII-hex output. Body content display is

truncated to a maximum of 4 KB with no output given for an empty body. Body content output follows the header output, for example:

```
...
GIOP Hdr (len 12): [47][49][4f][50][01][02][00][01][00][00][00][0c]
Msg Hdr  (len 12): [00][00][00][01][00][00][00][00][00][00][00][00]
Msg Body (len <x>): <content>
...
```

Configuring a CORBA Binding

CORBA bindings are described using a variety of Micro Focus-specific WSDL elements within the WSDL binding element. In most cases, the CORBA binding description is generated automatically using the wsdltocorba utility. Usually, it is unnecessary to modify generated CORBA bindings.

Namespace

The WSDL extensions used to describe CORBA data mappings and CORBA transport details are conventionally prefixed by the namespace prefix, `corba`. The definition of the `corba` namespace prefix is as follows:

```
xmlns:corba="http://schemas.iona.com/bindings/corba"
```

corba:binding element

The `corba:binding` element indicates that the binding is a CORBA binding. This element has one required attribute: `repositoryID`. `repositoryID` specifies the full type ID of the interface. The type ID is embedded in the object's IOR and therefore must conform to the IDs that are generated from an IDL compiler. These are of the form:

```
IDL:module/interface:1.0
```

The `corba:binding` element also has an optional attribute, `bases`, that specifies that the interface being bound inherits from another interface. The value for `bases` is the type ID of the interface from which the bound interface inherits. For example, the following IDL:

```
//IDL
interface clash{};
interface bad : clash{};
```

would produce the following `corba:binding`:

```
<corba:binding repositoryID="IDL:bad:1.0"
               bases="IDL:clash:1.0"/>
```

corba:operation element

The `corba:operation` element is a -specific element of `<operation>` and describes the parts of the operation's messages. `<corba:operation>` takes a single attribute, `name`, which duplicates the name given in `<operation>`.

corba:param element

The corba:param element is a member of <corba:operation>. Each <part> of the input and output messages specified in the logical operation, except for the part representing the return value of the operation, must have a corresponding <corba:param>. The parameter order defined in the binding must match the order specified in the IDL definition of the operation. <corba:param> has the following required attributes:

mode	Specifies the direction of the parameter. The values directly correspond to the IDL directions: in, inout, out. Parameters set to in must be included in the input message of the logical operation. Parameters set to out must be included in the output message of the logical operation. Parameters set to inout must appear in both the input and output messages of the logical operation.
idltype	Specifies the IDL type of the parameter. The type names are prefaced with corba: for primitive IDL types, and corbatm: for complex data types, which are mapped out in the corba:typeMapping portion of the contract.
name	Specifies the name of the parameter as given in the logical portion of the contract.

corba:return element

The corba:return element is a member of <corba:operation> and specifies the return type, if any, of the operation. It only has two attributes:

name	Specifies the name of the parameter as given in the logical portion of the contract.
idltype	Specifies the IDL type of the parameter. The type names are prefaced with corba: for primitive IDL types and corbatm: for complex data types which are mapped out in the corba:typeMapping portion of the contract.

corba:raises element

The corba:raises element is a member of <corba:operation> and describes any exceptions the operation can raise. The exceptions are defined as fault messages in the logical definition of the operation. Each fault message must have a corresponding corba:raises element. The corba:raises element has one required attribute, exception, which specifies the type of data returned in the exception.

In addition to operations specified in `<corba:operation>` tags, within the `<operation>` block, each `<operation>` in the binding must also specify empty `input` and `output` elements as required by the WSDL specification. The CORBA binding specification, however, does not use them.

For each fault message defined in the logical description of the operation, a corresponding `fault` element must be provided in the `<operation>`, as required by the WSDL specification. The `name` attribute of the `fault` element specifies the name of the schema type representing the data passed in the fault message.

Example

For example, a logical interface for a system to retrieve employee information might look similar to `personalInfoLookup`, shown in Example 54.

Example 54: *personalInfo lookup port type*

```
<message name="personalLookupRequest">
  <part name="empId" type="xsd:int" />
<message />
<message name="personalLookupResponse">
  <part name="return" element="xsd1:personalInfo" />
<message />
<message name="idNotFoundException">
  <part name="exception" element="xsd1:idNotFound" />
<message />
<portType name="personalInfoLookup">
  <operation name="lookup">
    <input name="empID" message="personalLookupRequest" />
    <output name="return" message="personalLookupResponse" />
    <fault name="exception" message="idNotFoundException" />
  </ operation>
</ portType>
```

The CORBA binding for `personalInfoLookup` is shown in Example 55.

Example 55: *personalInfoLookup CORBA Binding*

```
<binding name="personalInfoLookupBinding" type="tns:personalInfoLookup">
  <corba:binding repositoryID="IDL:personalInfoLookup:1.0"/>
  <operation name="lookup">
    <corba:operation name="lookup">
      <corba:param name="empId" mode="in" idltype="corba:long"/>
      <corba:return name="return" idltype="corbatm:personalInfo"/>
      <corba:raises exception="corbatm:idNotFound"/>
    </corba:operation>
    <input/>
    <output/>
    <fault name="personalInfoLookup.idNotFound"/>
  </operation>
</binding>
```

Configuring a CORBA Port

CORBA ports are described using the Micro Focus-specific WSDL elements, corba:address and corba:policy, within the WSDL port element, to specify how a CORBA object is exposed.

Namespace

The WSDL extensions used to describe CORBA data mappings and CORBA transport details are conventionally prefixed by the namespace prefix, `corba`. The definition of the `corba` namespace prefix is as follows:

```
xmlns:corba="http://schemas.iona.com/bindings/corba"
```

corba:address element

The IOR of the CORBA object is specified using the `corba:address` element. You have four options for specifying IORs in Artix contracts:

- Specify the objects IOR directly, by entering the object's IOR directly into the contract using the stringified IOR format:

  ```
  IOR:22342....
  ```

- Specify a file location for the IOR, using the following syntax:

  ```
  file:///file_name
  ```

Note: The file specification requires three backslashes (`///`).

It is usually simplest to specify the file name using an absolute path. If you specify the file name using a relative path, the location is taken to be relative to the directory the Artix process is started in, *not* relative to the containing WSDL file.

- Specify that the IOR is published to a CORBA name service, by entering the object's name using the `corbaname` format:

  ```
  corbaname:rir/NameService#object_name
  ```

  For more information on using the name service with Artix see *Deploying and Managing Artix Solutions*.

- Specify the IOR using `corbaloc`, by specifying the port at which the service exposes itself, using the `corbaloc` syntax.

  ```
  corbaloc:iiop:host:port/service_name
  ```

  When using `corbaloc`, you must be sure to configure your service to start up on the specified host and port.

corba:policy element

Using the optional `corba:policy` element, you can describe a number of POA polices the Artix service will use when creating the POA for connecting to a CORBA application. These policies include:

- POA Name.
- Persistence.
- ID Assignment.

Setting these policies lets you exploit some of the enterprise features of Micro Focus's Orbix 6.x, such as load balancing and fault tolerance, when deploying an Artix integration project. For information on using these advanced CORBA features, see the Orbix documentation.

POA Name

By default, an Artix POA is created with the default name, {*ServiceNamespace*}*ServiceLocalPart*#*PortName*. For example, if a CORBA port is defined by the following WSDL fragment:

```
<definitions
  ...
  xmlns:corbatm="http://iona.com/mycorbaservice" >

    <service name="CorbaService">
        <port binding="corbatm:CorbaBinding"
   name="CorbaPort">
            <corba:address

   location="file:../../hello_world_service.ior"/>
        </port>
    </service>
    ...
```

The unique POA name automatically generated for this CORBA port is {http://iona.com/mycorbaservice}CorbaService#CorbaPort.

Alternatively, you can specify the POA name explicitly by setting the `poaname` attribute, as follows:

```
<corba:policy poaname="poa_name" />
```

When setting a POA name using the `poaname` attribute, it is your responsibility to ensure that the POA name is unique. That is, the POA name should *not* be shared between CORBA ports within a service or across CORBA services.

Persistence

By default Artix POA's have a persistence policy of `false`. To set the POA's persistence policy to `true`, use the following:

```
<corba:policy persistent="true" />
```

ID Assignment

By default Artix POAs are created with a `SYSTEM_ID` policy, meaning that their ID is assigned by the ORB. To specify that the POA connecting a specific object should use a user-assigned ID, use the following:

```
<corba:policy serviceid="POAid" />
```

This creates a POA with a `USER_ID` policy and an object id of *`POAid`*.

Example

For example, a CORBA port for the `personalInfoLookup` binding would look similar to Example 56:

Example 56: *CORBA personalInfoLookup Port*

```
<service name="personalInfoLookupService">
  <port name="personalInfoLookupPort"
        binding="tns:personalInfoLookupBinding">
    <corba:address location="file:///objref.ior" />
    <corba:policy persistent="true" />
    <corba:policy serviceid="personalInfoLookup" />
  </ port>
</ service>
```

Artix expects the IOR for the CORBA object to be located in a file called `objref.ior` (relative to the directory in which the Artix process is started), and creates a persistent POA with an object id of `personalInfo` to connect the CORBA application.

CORBA Utilities in Artix

Use the idltowsdl utility to convert OMG IDL to WSDL and use the wsdltocorba utility to generate CORBA bindings and to convert WSDL to OMG IDL.

Generating a CORBA Binding

The `wsdltocorba` utility can perform two distinct tasks:

- Generate a CORBA binding.
- Convert WSDL to OMG IDL.

This section discusses how to use the `wsdltocorba` utility to add a CORBA binding to an existing WSDL contract.

Location

The `wsdltocorba` utility is located in the following `bin` directory:

- *ArtixInstallDir*`/bin`.

WSDLTOCORBA / WSDLTOIDL

C++ Runtime Utility

```
wsdltocorba -corba -i port-type [-d directory] [-o file]
   [-props namespace] [-?] [-v] [-verbose] wsdl_file
```

Options

The command has the following options:

`-corba`	Instructs the tool to generate a CORBA binding for the specified port type.
`-i` *`port-type`*	Specifies the name of the port type being mapped to a CORBA binding.
`-d` *`directory`*	Specifies the directory into which the new WSDL file is written.
`-o` *`file`*	Specifies the name of the generated WSDL file. Defaults to *`wsdl_file`*`-corba.wsdl`.
`-props` *`namespace`*	Specifies the target namespace for the `corba:typeMapping` element (an element that defines the WSDL-to-IDL mappings for complex types).
`-?`	Display detailed information about the options.
`-v`	Display the version of the utility.
`-verbose`	Write a detailed log to standard output while the utility is running.

Converting WSDL to OMG IDL

The `wsdltocorba` utility can perform two distinct tasks:

- Generate a CORBA binding.
- Convert WSDL to OMG IDL.

This section discusses how to use the `wsdltocorba` utility to convert a WSDL contract into an OMG IDL file.

Location

The `wsdltocorba` utility is located in the following `bin` directory:

- *ArtixInstallDir*`/bin`.

WSDLTOCORBA / WSDLTOIDL

C++ Runtime

```
wsdltocorba -idl -b binding [-d directory] [-o file] [-?] [-v]
   [-verbose] wsdl_file
```

Options

The command has the following options:

`-idl`	Instructs the tool to generate an IDL file from the specified binding.
`-b` *`binding`*	Specifies the CORBA binding from which to generate IDL.
`-d` *`directory`*	Specifies the directory into which the new IDL file is written.
`-o` *`file`*	Specifies the name of the generated IDL file. Defaults to *`wsdl_file`*`.idl`.
`-?`	Display detailed information about the options.
`-v`	Display the version of the utility.
`-verbose`	Write a detailed log to standard output while the utility is running.

Converting OMG IDL to WSDL

Micro Focus's IDL compiler supports several command line flags that specify how to create a WSDL file from an IDL file. The default behavior of the tool is to create WSDL file that uses wrapped doc/literal style messages. Wrapped doc/literal style messages have a single part, defined using an element that wraps all of the elements in the message.

Location

The `idltowsdl` utility is located in the following `bin` directory:

- *ArtixInstallDir*`/bin`.

IDLTOWSDL

C++ Runtime

```
idltowsdl [ -I idl-include-directory ]* [ -3 ] [ -o output-directory
   ] [ -a corba-address ] [ -b ] [ -f corba-address-file ] [ -n
   schema-import-file ] [ -s idl-sequence-type ] [ -w target-namespace
   ] [ -x schema-namespace ] [ -t type-map-namespace ] [ -useTypes ]
   [ -unwrap ] [ -r reference-schema-file ] [ -L logical-wsdl-file ] [
   -P physical-wsdl-file ] [ -T schema-file-name ] [ -fasttrack ] [
   -interface interface-name ] [ -soapaddr soap-port-address ] [
   -qualified ] [ -inline ] [ -e xml-encoding-type ] [ -? ] [ -v ] [
   -verbose ] IDLFile
```

Options

The command has the following options:

`-I` *`idl-include-directory`*	Specify a directory to be included in the search path for the IDL preprocessor.
`-3`	Select parsing mode for compatibility with legacy Orbix 3 IDL files.
`-o` *`output-directory`*	Specifies the directory into which the WSDL file is written.
`-a` *`corba-address`*	Specifies an absolute address through which the object reference may be accessed. The *`corba-address`* may be a relative or absolute path to a file, or a corbaname URL
`-b`	Specifies that bounded strings are to be treated as unbounded. This eliminates the generation of the special types for the bounded string.
`-f` *`corba-address-file`*	Specifies a file containing a string representation of an object reference. The object reference is placed in the `corba:address` element in the `<port>` definition of the generated service. The *`corba-address-file`* must exist when you run the `idltowsdl` utility.
`-n` *`schema-import-file`*	Specifies that a schema file, *`schema-import-file`*, is to be included in the generated contract by an import statement. This option cannot be used with the `-T` option.
`-s` *`idl-sequence-type`*	Specifies the XML schema type used to map the IDL `sequence<octet>` type. Valid values are `base64Binary` or `hexBinary`. The default is `base64Binary`.
`-w` *`target-namespace`*	Specifies the namespace to use for the WSDL `targetNamespace`. The default is `http://schemas.iona.com/idl/`*`IDLFile`*.
`-x` *`schema-namespace`*	Specifies the namespace to use for the Schema `targetNamespace`. The default is `http://schemas.iona.com/idltypes/`*`IDLFile`*.

`-t` *`type-map-namespace`*	Specifies the namespace to use for the CORBA TypeMapping `targetNamespace`. The default is `http://schemas.iona.com/typemap/corba/`*`IDL File`*.
`-useTypes`	Generate rpc style messages. rpc style messages have parts defined using XMLSchema types instead of XML elements.
`-unwrap`	Generate unwrapped doc/literal messages. Unwrapped messages have parts that represent individual elements. Unlike wrapped messages, unwrapped messages can have multiple parts and are not allowed by the WS-I.
`-r` *`reference-schema-file`*	Specify the pathname of the schema file imported to define the `wsa:EndpointReference` type. If the `-r` option is not given, the idl compiler gets the schema file pathname from the `AddressingSchemaLocation` setting in `etc/idl.cfg`.
`-L` *`logical-wsdl-file`*	Specifies that the logical portion of the generated WSDL specification into is written to *`logical-wsdl-file`*. The *`logical-wsdl-file`* is then imported into the default generated file.
`-P` *`physical-wsdl-filen`*	Specifies that the physical portion of the generated WSDL specification into is written to *`physical-wsdl-file`*. The *`physical-wsdl-file`* is then imported into the default generated file.
`-T` *`schema-file-name`*	Specifies that the schema types are to be generated into a separate file. The schema file is included in the generated contract using an import statement. This option cannot be used with the `-n` option.
`-fasttrack`	Provides a fast way of generating a router contract for a router that converts incoming SOAP/HTTP messages into CORBA invocations. The `-interface` option must always be specified when `-fasttrack` is used.
`-interface` *`interface-name`*	Used in combination with the `-fasttrack` option to specify the IDL interface that is exposed through the generated router contract.
`-soapaddr` *`soap-port-address`*	Used in combination with the `-fasttrack` option to specify the address of the generated SOAP port. The address is specified in the format *`Host`*`:`*`Port`*.

`-qualified`	Generate the schemas in the WSDL contract with the `elementFormDefault` and `attributeFormDefault` attributes set to `qualified`. This implies that elements and attributes appearing in instance documents must be explicitly qualified by a namespace.
`-inline`	Normally, when you specify a schema file using the `-n` option, the schema is imported by a generated `xsd:import` element, which sets the `schemaLocation` attribute. If you specify the `-inline` option, however, the schema is included directly in the generated WSDL contract and the generated `xsd:import` element omits the `schemaLocation` attribute.
`-e` *`xml-encoding-type`*	Use the specified WSDL encoding for the value of the `encoding` attribute in the generated `<?xml ... ?>` tag. The default is UTF-8.
`-?`	Display detailed information about the options.
`-v`	Display the version of the utility.
`-verbose`	Write a detailed log to standard output while the utility is running.

Note: The command line flag entries are case sensitive even on Windows. Capitalization in your generated WSDL file must match the capitalization used in the prewritten code.

Orbix 3 legacy compatibility

To address some issues associated with Orbix 3 migration, the Artix IDL compiler supports a `-3` option, which causes the following behavior in the `idltowsdl` utility:

- Case sensitivity is activated—this means that name lookup during parsing is case sensitive. While technically incorrect according to the CORBA specification, some legacy IDL files might require case sensitivity. The IDL compiler issues warnings, if case sensitivity rules are broken.
- New IDL keywords added since CORBA 2.3 (for example, `factory` and `local`) are treated as ordinary identifiers, but warnings are issued.
- If a different spelling of the keyword `Object` is encountered (for example, `object`, `OBJECT`, or `oBjEcT`), it is treated as an identifier, and a warning is issued.
- All IDL is preprocessed with the additional flag `-DIT_ORBIX3IDL_COMPATIBILITY`. This allows IDL definitions to make use of this macro in `#ifdef`s to help with migration issues.

- Unscoped types from the `CORBA` module—legacy IDL often uses `TypeCode` as a global type, whereas the IDL specification requires it to be properly scoped to the `CORBA` module. To deal with this issue, you could use the following `#ifdef` to bring `TypeCode` into global scope, if required:

  ```
  #ifdef IT_ORBIX3IDL_COMPATIBILITY
  typedef CORBA::TypeCode TypeCode;
  #endif
  ```

  Note: `TypeCode` originally was a global type in CORBA, but the `CORBA` module was added around 1992/1993 to scope such types.)

- Semicolons are tolerated in `#include` statements. The IDL compiler removes the semicolons and issues a warning.
- Opaque types—there are no easy migration solutions for opaque types. The IDL compiler does not recognize the `opaque` keyword. If you have legacy IDL that uses opaque types, you should consider migrating them to something like a `valuetype` instead.

Mapping CORBA Exceptions

To facilitate interoperability between CORBA applications and Artix applications, Artix automatically maps between CORBA system exceptions and Artix faults.

Mapping from CORBA System Exceptions

When a CORBA system exception is returned from a CORBA server to an Artix client, Artix automatically converts the CORBA system exception to a fault category.

Map from CORBA system exceptions to fault categories

Table 5 shows how each of the major CORBA system exceptions map to Artix fault categories.

Table 5: *Map from CORBA System Exceptions to Fault Categories*

CORBA System Exception	Fault Category
CORBA::BAD_CONTEXT	IT_Bus::FaultCategory::INTERNAL
CORBA::BAD_INV_ORDER	IT_Bus::FaultCategory::INTERNAL
CORBA::BAD_OPERATION	IT_Bus::FaultCategory::BAD_OPERATION
CORBA::BAD_TYPECODE	IT_Bus::FaultCategory::MARSHAL_ERROR
CORBA::BAD_QOS	IT_Bus::FaultCategory::INTERNAL
CORBA::CODESET_INCOMPATIBLE	IT_Bus::FaultCategory::MARSHAL_ERROR
CORBA::COMM_FAILURE	IT_Bus::FaultCategory::CONNECTION_FAILURE
CORBA::DATA_CONVERSION	IT_Bus::FaultCategory::MARSHAL_ERROR
CORBA::FREE_MEM	IT_Bus::FaultCategory::MEMORY
CORBA::IMP_LIMIT	IT_Bus::FaultCategory::INTERNAL
CORBA::INITIALIZE	IT_Bus::FaultCategory::UNKNOWN
CORBA::INTERNAL	IT_Bus::FaultCategory::INTERNAL
CORBA::INTF_REPOS	IT_Bus::FaultCategory::INTERNAL
CORBA::INV_FLAG	IT_Bus::FaultCategory::INTERNAL
CORBA::INV_IDENT	IT_Bus::FaultCategory::NOT_EXIST
CORBA::INV_OBJREF	IT_Bus::FaultCategory::INVALID_REFERENCE
CORBA::INV_POLICY	IT_Bus::FaultCategory::INTERNAL

Table 5: *Map from CORBA System Exceptions to Fault Categories*

CORBA System Exception	Fault Category
CORBA::INVALID_TRANSACTION	IT_Bus::FaultCategory::INTERNAL
CORBA::MARSHAL	IT_Bus::FaultCategory::MARSHAL_ERROR
CORBA::NO_IMPLEMENT	IT_Bus::FaultCategory::NOT_IMPLEMENTED
CORBA::NO_MEMORY	IT_Bus::FaultCategory::MEMORY
CORBA::NO_PERMISSION	IT_Bus::FaultCategory::NO_PERMISSION
CORBA::NO_RESOURCES	IT_Bus::FaultCategory::INTERNAL
CORBA::NO_RESPONSE	IT_Bus::FaultCategory::INTERNAL
CORBA::OBJ_ADAPTER	IT_Bus::FaultCategory::INTERNAL
CORBA::OBJECT_NOT_EXIST	IT_Bus::FaultCategory::NOT_EXIST
CORBA::PERSIST_STORE	IT_Bus::FaultCategory::INTERNAL
CORBA::REBIND	IT_Bus::FaultCategory::INTERNAL
CORBA::TIMEOUT	IT_Bus::FaultCategory::TIMEOUT
CORBA::TRANSACTION_MODE	IT_Bus::FaultCategory::INTERNAL
CORBA::TRANSACTION_REQUIRED	IT_Bus::FaultCategory::INTERNAL
CORBA::TRANSACTION_ROLLEDBACK	IT_Bus::FaultCategory::INTERNAL
CORBA::TRANSACTION_UNAVAILABLE	IT_Bus::FaultCategory::INTERNAL
CORBA::TRANSIENT	IT_Bus::FaultCategory::TRANSIENT

Mapping from Fault Categories

When a fault (that is, a built-in exception) is returned from an Artix server to a CORBA client, Artix automatically converts the fault category to a CORBA system exception.

Map from CORBA system exceptions to fault categories

Table 6 shows how each of the Artix fault categories map to major CORBA system exceptions.

Table 6: *Map from CORBA System Exceptions to Fault Categories*

Fault Category	CORBA System Exception
IT_Bus::FaultCategory::BAD_OPERATION	CORBA::BAD_OPERATION
IT_Bus::FaultCategory::CONNECTION_FAILURE	CORBA::COMM_FAILURE
IT_Bus::FaultCategory::INTERNAL	CORBA::INTERNAL
IT_Bus::FaultCategory::INVALID_REFERENCE	CORBA::INV_OBJREF

Table 6: *Map from CORBA System Exceptions to Fault Categories*

Fault Category	CORBA System Exception
IT_Bus::FaultCategory::LICENSE	CORBA::NO_IMPLEMENT
IT_Bus::FaultCategory::MARSHAL_ERROR	CORBA::MARSHAL
IT_Bus::FaultCategory::MEMORY	CORBA::NO_MEMORY
IT_Bus::FaultCategory::NO_PERMISSION	CORBA::NO_PERMISSION
IT_Bus::FaultCategory::NOT_EXIST	CORBA::OBJECT_NOT_EXIST
IT_Bus::FaultCategory::NOT_IMPLEMENTED	CORBA::NO_IMPLEMENT
IT_Bus::FaultCategory::NOT_UNDERSTOOD	CORBA::BAD_PARAM
IT_Bus::FaultCategory::TIMEOUT	CORBA::TIMEOUT
IT_Bus::FaultCategory::TRANSIENT	CORBA::TRANSIENT
IT_Bus::FaultCategory::UNKNOWN	CORBA::INITIALIZE
IT_Bus::FaultCategory::VERSION_ERROR	CORBA::BAD_PARAM

Mapping of Completion Status

The CORBA completion status flag and the Artix fault completion status flag have exactly the same semantics and are thus effectively equivalent. In other words, a YES completion status implies that the remote operation completed its work; a NO completion status implies that the remote operation was never called; and a MAYBE completion status implies that it is impossible to say whether or not the remote operation completed its work.

Completion status mapping

Table 7 shows the mapping between CORBA completion status values and fault completion status values.

Table 7: *Completion Status Mapping*

CORBA Completion Status	Fault Completion Status
CORBA::COMPLETED_YES	IT_Bus::FaultCompletionStatus::YES
CORBA::COMPLETED_NO	IT_Bus::FaultCompletionStatus::NO
CORBA::COMPLETED_MAYBE	IT_Bus::FaultCompletionStatus::MAYBE

Index

L

M

N

O

P

Q

R

S

T

U

W

X