Artix 5.6.3

Developing Artix Applications in C++

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

http://www.microfocus.com

2015-02-18

Contents

Preface

What is Covered in This Book

This book covers the information needed to develop applications using the Artix C++ API.

Who Should Read This Book

This guide is intended for Artix C++ programmers. In addition to a knowledge of C++, this guide assumes that the reader is familiar with WSDL and XML schemas.

The Artix Documentation Library

For information on the entire Artix Documentation Library, including organization, contents, conventions, and reading paths, 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

Getting Started with Artix Programming

This chapter shows you how to rapidly build and deploy a complete client/server application using the Artix command-line tools.

The Hello World Application

Figure 1 provides a brief overview of the Hello World application, a simple two-tier client/server application, on which the rest of this chapter is based. The communication protocol for this example is SOAP over HTTP.

The server exposes a service, `HelloWorldSOAPService`, which listens on a single HTTP port for incoming invocation requests.

The client obtains the connection details for the `HelloWorldSOAPService` by reading a local copy of the Hello World WSDL contract. The client then calls the two operations, `sayHi` and `greetMe`, that are supported by the Hello World service.

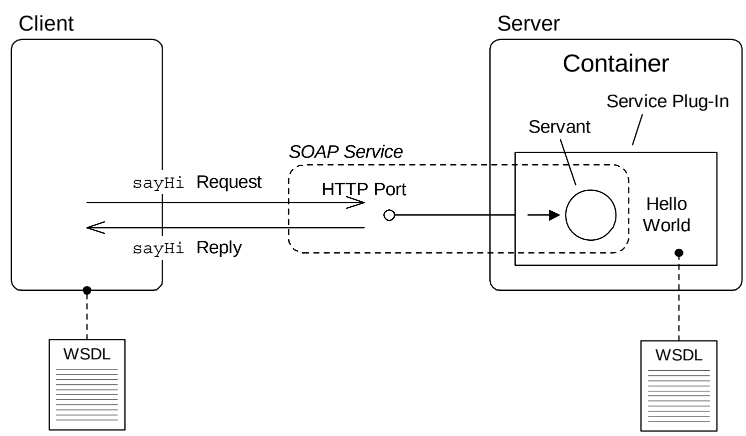

Figure 1: *The Hello World Application*

WSDL contract

The Web Services Description Language (WSDL) contract provides the foundation for the Hello World distributed application. The contract contains all of the information needed by a Web services client, including a detailed description of the Hello World Web service and details of the operations supported by the service. The WSDL contract contains the following main sections:

- *WSDL port type*—describes the interface for the Hello World service, including all of the WSDL operations supported by the service. The Hello World port type is named `Greeter` and contains the following operations:
    - `sayHi`—requests the server to send a message of greeting (the operation returns a string).
    - `greetMe`—sends the user's name to the server and requests the server to send a personalized greeting (the operation takes a single string argument and returns a string).
- *WSDL binding*—describes how operation request and reply messages are to be encoded. For example, the Hello World application encodes messages in a SOAP format.

  Artix provides tools to generate the WSDL binding automatically.
- *WSDL service and port*—provides connection data and properties for a particular transport. For transports based on the Internet Protocol, you can specify the service's hostname and IP port. For example, the Hello World service uses the HTTP transport and the connection data is specified in the form of a HTTP URL.

Server

The server provides the implementation of the Hello World Web service. In particular, it provides a servant class that implements the `sayHi` and `greetMe` WSDL operations.

The preferred approach for building and deploying an Artix server is to use the *container model*. The Artix container model is based on the idea that the server can be broken up into the following parts:

- Artix container.
- Service plug-in.

Artix container

The Artix container is an executable, `it_container`, that provides a basic environment for Web services to run in. Service implementations are loaded into the container as plug-ins. Artix exploits the dynamic loading capabilities of modern operating systems to load service plug-ins as shared libraries or DLLs.

Service plug-in

A *service plug-in* is an Artix plug-in that contains the implementation of one or more servant classes. Typically, a servant class is responsible for implementing the operations from a single WSDL port type. Implementing a servant class in C++ is equivalent to implementing a Web service.

Client

The client is a standalone executable that invokes the `sayHi` and `greetMe` operations from the Hello World service.

The key artifact on the client side is the *client proxy* class, which provides an interface mapped from the `Greeter` port type. By calling functions on a client proxy object, a client can initiate remote procedure calls on the corresponding operations in the remote Web service.

Prerequisites

Before attempting to build and run the Hello World application, check that the following prerequisites are satisfied:

- Artix environment script.
- C++ compiler.

Artix environment script

Artix provides a script, `artix_env.bat` or `artix_env.sh`, in *ArtixInstallDir*/`bin`, that sets a variety of environment variables (not just the basic ones mentioned here). If your user account is not configured to run this script, you might have to run it manually.

Depending on what compiler you use and what platform you are running on, it might be necessary to run the `artix_env` script with particular command-line switches. For details, see the *Artix Installation Guide*.

C++ compiler

Make sure that your environment is configured to use the correct version of C++ compiler. In general, it is necessary to use *precisely* the right compiler version, as specified in the *Artix Installation Guide*.

Define a WSDL Contract

This section assumes that you already have the logical part of the contract (that is, the WSDL port type and associated type definitions) and shows you how to proceed to generate the rest of the contract (WSDL binding and WSDL service) using the Artix command-line tools. In particular, this section describes how to define a WSDL contract for the Hello World application.

To define a Hello World WSDL contract, perform the following steps:

1. Example directories.
2. Define the logical contract.
3. Add a SOAP binding to the contract.
4. Add a HTTP endpoint to the contract.

Example directories

First of all, you need to create a few directories to hold the files associated with the Hello World example. In a convenient location of your choosing, create the following directories:

ArtixExampleDir
ArtixExampleDir/etc
ArtixExampleDir/client
ArtixExampleDir/server

Where *ArtixExampleDir* is the root of your example directory tree.

Define the logical contract

The logical part of a WSDL contract is the part that contains the *WSDL port type* definitions, along with the requisite definitions of any associated message types and XML schema types.

If you are defining a logical contract from scratch, you can write the contract directly (assuming you are sufficiently familiar with the syntax for XML schemas and WSDL contracts). For the Hello World example, use the logical contract from Example 1.

Example 1: *Logical Contract for the Hello World Example*

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="HelloWorld"
   targetNamespace="http://www.iona.com/hello_world_soap_http"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.iona.com/hello_world_soap_http"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:types>
        <schema targetNamespace="http://www.iona.com/hello_world_soap_http"
            xmlns="http://www.w3.org/2001/XMLSchema">
            <element name="responseType" type="xsd:string"/>
            <element name="requestType" type="xsd:string"/>
        </schema>
    </wsdl:types>
    <wsdl:message name="sayHiRequest"/>
    <wsdl:message name="sayHiResponse">
        <wsdl:part element="tns:responseType" name="theResponse"/>
    </wsdl:message>
    <wsdl:message name="greetMeRequest">
        <wsdl:part element="tns:requestType" name="me"/>
    </wsdl:message>
    <wsdl:message name="greetMeResponse">
```

Example 1: *Logical Contract for the Hello World Example*

```
        <wsdl:part element="tns:responseType" name="theResponse"/>
    </wsdl:message>

    <wsdl:portType name="Greeter">
        <wsdl:operation name="sayHi">
            <wsdl:input message="tns:sayHiRequest" name="sayHiRequest"/>
            <wsdl:output message="tns:sayHiResponse" name="sayHiResponse"/>
        </wsdl:operation>
        <wsdl:operation name="greetMe">
            <wsdl:input message="tns:greetMeRequest" name="greetMeRequest"/>
            <wsdl:output message="tns:greetMeResponse" name="greetMeResponse"/>
        </wsdl:operation>
    </wsdl:portType>
</wsdl:definitions>
```

Where the Hello World contract defines a single port type, `Greeter`, having two operations, `sayHi()` and `greetMe()`. The `sayHi()` operation returns a string. The `greetMe()` operation takes a single string argument and returns a string.

Using your favorite text editor, copy the WSDL contract from Example 1 on page 6 into the following file:

ArtixExampleDir`/etc/_hello_world.wsdl`

Add a SOAP binding to the contract

The SOAP binding describes the encoding of request and reply messages in the SOAP protocol. By adding a SOAP binding for the `Greeter` port type from Example 1 on page 6, you make it possible to invoke `Greeter`'s operations using a SOAP protocol. Note that the SOAP binding describes only how the messages are encoded, it does *not* describe how to send the messages to and from the remote service (that is the responsibility of the transport).

To add a SOAP binding to the contract, change directory to *ArtixExampleDir*`/etc` and enter the following command:

```
wsdltosoap -i Greeter
   -b GreeterSOAPBinding
   _hello_world.wsdl
```

In this example, the `wsdltosoap` command takes the following switches:

`-i` *PortType*	WSDL port type for which to generate a binding.
`-b` *Binding*	Name of the newly generated binding.

This command generates a new file, `_hello_world-soap.wsdl`, which contains the SOAP binding shown in Example 2.

Example 2: *SOAP Binding for the Greeter Port Type*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >
    ...
```

Example 2: *SOAP Binding for the Greeter Port Type*

```
    <binding name="GreeterSOAPBinding" type="tns:Greeter">
        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="sayHi">
            <soap:operation soapAction="" style="document"/>
            <input name="sayHiRequest">
                <soap:body use="literal"/>
            </input>
            <output name="sayHiResponse">
                <soap:body use="literal"/>
            </output>
        </operation>
        <operation name="greetMe">
            <soap:operation soapAction="" style="document"/>
            <input name="greetMeRequest">
                <soap:body use="literal"/>
            </input>
            <output name="greetMeResponse">
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>
</definitions>
```

Add a HTTP endpoint to the contract

To enable you to invoke `Greeter`'s operations over SOAP/HTTP, you must add a HTTP endpoint to the contract. A typical HTTP endpoint consists of a `service` element containing a single `port` element. In the `port` element, you can indicate that the transport protocol is HTTP and you can provide the relevant properties for the HTTP endpoint.

To add a HTTP endpoint to the contract, change directory to *ArtixExampleDir*`/etc` and enter the following command:

```
wsdltoservice -b GreeterSOAPBinding
   -e HelloWorldSOAPService
   -t HTTPPort
   -transport http
   -a http://localhost:4444
   -o hello_world.wsdl
   _hello_world-soap.wsdl
```

In this example, the `wsdltoservice` command takes the following switches:

`-b` *Binding*	Binding for which an endpoint is to be generated.
`-e` *ServiceName*	The name of the new WSDL service.
`-t` *PortName*	The name of the new WSDL port.
`-transport http`	Specifies that this is a HTTP endpoint.
`-a` *LocationURL*	The location URL for the new endpoint.
`-o` *OutputFile*	The name of the output file containing the updated WSDL contract.

This command generates a new file, `hello_world.wsdl`, which contains the HTTP endpoint shown in Example 3.

Example 3: *HTTP Endpoint for the Greeter Port Type*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >
    ...
    <service name="HelloWorldSOAPService">
        <port binding="tns:GreeterSOAPBinding" name="HTTPPort">
            <http:address location="http://localhost:4444"/>
        </port>
    </service>
</definitions>
```

Develop a Service Plug-In

To develop a service plug-in for the Hello World WSDL application, perform the following steps:

1. Generate service code from the WSDL contract.
2. Edit the servant class.
3. Compile the service plug-in.

Generate service code from the WSDL contract

Artix has a built-in code generator that can automatically generate most of the code required for a simple service plug-in.

To generate service plug-in code from the Hello World WSDL contract, open a command prompt, change directory to *ArtixExampleDir*/`server`, and enter the following command (for your respective platform):

Windows

```
wsdltocpp -i Greeter
   -e HelloWorldSOAPService
   -server
   -impl
   -m NMAKE:library
   -plugin:it_hello_world
   -deployable
   ..\etc\hello_world.wsdl
```

UNIX

```
wsdltocpp -i Greeter
   -e HelloWorldSOAPService
   -server
   -impl
   -m UNIX:library
   -plugin:it_hello_world
   -deployable
   ../etc/hello_world.wsdl
```

In this example, the `wsdltocpp` command takes the following switches:

`-i` *PortType*	The port type for which code is to be generated.
`-e` *ServiceName*	The WSDL service associated with the port type.
`-server`	Generate server skeleton code.
`-impl`	Provide an outline implementation of the `Greeter` servant class.
`-m [NMAKE|UNIX]:library`	Generate a makefile that builds the service plug-in library (for Windows and UNIX respectively).
`-plugin:`*LibName*	Generate the code required for a plug-in library, using *LibName* as the root name of the library.
`-deployable`	Generate a deployment descriptor file for the service plug-in.

The preceding command generates all of the files needed to build and deploy the Hello World service plug-in. The plug-in is packaged in the form of a shared library or DLL.

Edit the servant class

The generated `GreeterImpl` servant class is the class that actually implements the `Greeter` port type. In order to implement the Hello World service, all that you need to do is to implement the relevant functions in this class. An outline implementation of the `GreeterImpl` class is provided in the `GreeterImpl.cxx` file.

To complete the implementation of the `GreeterImpl` servant class, open the `GreeterImpl.cxx` file with your favorite text editor and edit the `sayHi()` and `greetMe()` functions as shown in Example 4.

Example 4: *Sample Implementations of sayHi() and greetMe()*

```
// C++
...

void
GreeterImpl::sayHi(
    IT_Bus::String &theResponse
) IT_THROW_DECL((IT_Bus::Exception))
{
    std::cout << "GreeterImpl::sayHi() called." << std::endl;
    theResponse = "Greetings from the Artix HelloWorld service.";
}

void
GreeterImpl::greetMe(
    const IT_Bus::String &me,
    IT_Bus::String &theResponse
) IT_THROW_DECL((IT_Bus::Exception))
{
    std::cout << "GreeterImpl::greetMe() called." << std::endl;
```

Example 4: *Sample Implementations of sayHi() and greetMe()*

```
    theResponse = "Hello " + me;
}
```

Edit the `sayHi()` and `greetMe()` functions, replacing the function bodies with the lines of code highlighted in bold font.

Compile the service plug-in

To compile the service plug-in, enter the following at a command prompt:

Windows

`nmake all`

UNIX

`make all`

> **Note:** It is essential to specify `all` as the make target, because the default target does not generate the dependencies file.

Develop a Client

To develop a client for the Hello World WSDL application, perform the following steps:

1. Generate client code from the WSDL contract.
2. Edit the client main() function.
3. Compile the client.

Generate client code from the WSDL contract

To generate client code from the Hello World WSDL contract, open a command prompt, change directory to *ArtixExampleDir*/`client`, and enter the following command (for your respective platform):

Windows

```
wsdltocpp -i Greeter
   -e HelloWorldSOAPService
   -client
   -sample
   -m NMAKE:executable
   ..\etc\hello_world.wsdl
```

UNIX

```
wsdltocpp -i Greeter
   -e HelloWorldSOAPService
   -client
   -sample
   -m UNIX:executable
   ../etc/hello_world.wsdl
```

In this example, the `wsdltocpp` command takes the following switches:

`-i` *PortType*	The port type for which code is to be generated.
`-e` *ServiceName*	The WSDL service associated with the port type.
`-client`	Generate client stub code.
`-sample`	Provide an outline implementation of the client's `main()` function.
`-m [NMAKE|UNIX]:executable`	Generate a makefile that builds the client executable (for Windows and UNIX respectively).

The preceding command generates all of the files needed to build a client of the Hello World service. The client is implemented as a standalone executable.

Edit the client main() function

An outline implementation of the client `main()` function is provided in the generated `GreeterClientSample.cxx` file.

To complete the implementation of the client, open the `GreeterClientSample.cxx` file with your favorite text editor and edit the `main()` function as shown in Example 5, adding the lines of code shown in bold font.

Example 5: *Client main() function for Hello World Application*

```
// C++
...
try
{
    /*
     *  Create an instance of the web service client
     */
    IT_Bus::init(argc, argv);

    GreeterClient client;
    ...
    IT_Bus::String theResponse;
    client.sayHi(theResponse);
   cout << "sayHi() returned: \"" << theResponse << "\"" << endl;

    IT_Bus::String me = "YourName";
    client.greetMe(me, theResponse);
    cout << "greetMe() returned: \"" << theResponse << "\"" << endl;
}
catch(IT_Bus::Exception& e)
...
```

The additional lines of code invoke the `sayHi()` and `greetMe()` operations on the `HelloWorldSOAPService` service. The client code performs the following steps:

1. *Initialize an Artix Bus instance*—the call to `IT_Bus::init()` initializes an Artix Bus object (of `IT_Bus::Bus` type), which provides the basic Artix functionality.
2. *Create a client proxy instance*—a client proxy is an object that encapsulates the information required to contact a remote WSDL service. In this example, the `GreeterClient` class is the proxy for the `HelloWorldSOAPService` service.

   If you call the default constructor (as here), the client proxy is constructed with default values for the WSDL contract location, service name, and port name (the defaults are hard-coded in the client stub file, `GreeterClient.cxx`).
3. *Invoke the* `sayHi()` *and* `greetMe()` *operations on the remote* `HelloWorldSOAPService` *service*—you can invoke the remote `Greeter` operations by calling the `sayHi()` and `greetMe()` operations on the client proxy, `client`.

Compile the client

To compile the service plug-in, enter the following at a command prompt:

Windows

```
nmake all
```

UNIX

```
make all
```

Run the Application

To run the Hello World WSDL application, perform the following steps:

1. Run the container and load the service plug-in.
2. Run the client.

Run the container and load the service plug-in

To run the container and load the Hello World service plug-in, open a command prompt, change directory to *ArtixExampleDir*/`server`, and enter the following command:

```
it_container -publish -deploy deployHelloWorldSOAPService.xml
```

After issuing this command, the Artix container starts up and the `HelloWorldSOAPService` is activated. You should see the following output logged to the console screen:

```
Micro Focus it_container server starting
Micro Focus it_container server ready
```

See ***Configuring and Deploying Artix Solutions*** for more details on running the Artix container.

Run the client

To run the sample client, open a command prompt, change directory to *ArtixExampleDir*/`client`, and enter the following command:

```
GreeterClient
```

You should see the following output logged to the console screen:

```
 GreeterClient
sayHi() returned: "Greetings from the Artix HelloWorld service."
greetMe() returned: "Hello YourName"
```

Adding Configuration to the Application

The Artix configuration file

The Artix configuration file, *ArtixConfig*`.cfg`, is a local file that contains configuration settings for Artix applications. It is primarily used for settings that do not belong in a WSDL contract (although there is some overlap between WSDL contract settings and Artix configuration file settings).

For more details about Artix configuration files, see *Configuring and Deploying Artix Solutions*.

Default configuration file

Artix provides a default configuration file, which is located in the *ArtixInstallDir*/`etc/domains` directory. This is `artix.cfg`, and Artix is set up to use this configuration file by default.

Sample configuration for Hello World

Example 6 shows an example of a configuration file that can be used for the Hello World application.

Example 6: *Sample Configuration for the Hello World Application*

```
  # Artix Configuration File
1 include "ArtixInstallDir\etc\domains\artix.cfg";

  artix_example {
2     client {
3         orb_plugins = ["xmlfile_log_stream"];
      };

4     server {
          orb_plugins = ["xmlfile_log_stream"];
5         bus:initial_contract_dir = ["ArtixExampleDir\etc"];
      };
  };
```

The preceding configuration can be described as follows:

1. The `artix.cfg` file is the default configuration file provided with Artix. It contains many default configuration settings, which are needed by all Artix applications.

   You should include the `artix.cfg` file in your own Artix configuration file by invoking the `include` directive, as shown. You need to edit the pathname from this example to match the actual location of `artix.cfg` in your Artix installation.
2. The configuration scope, `artix_example.client`, contains the settings specific to the Hello World client.
3. The `orb_plugins` list specifies the set of Artix plug-ins to load at program start-up time. Additional plug-ins can be loaded later on, if needed, through the dynamic loading capability of the Artix plug-in framework.

   In the current example, just the XML logging plug-in, `xmlfile_log_stream`, is loaded at program start-up time.

   > **Note:** The majority of Artix plug-ins are loaded dynamically, in the course of parsing a WSDL contract.
   >
   > For example, if a WSDL contract has a port that uses the HTTP transport protocol, Artix automatically loads the `at_http` plug-in to enable support for HTTP.

4. The configuration scope, `artix_example.server`, contains the settings specific to the Hello World service plug-in.
5. The `bus:initial_contract_dir` configuration variable gives the location of a directory containing WSDL contracts. Artix searches this directory to locate the service plug-in's WSDL contract.

   Artix provides a variety of other ways to specify the location of the service's WSDL contract—for more details, see "Options for providing WSDL contracts" on page 51.

Command-line switches for configuration

To run an Artix program with a configuration other than the default, you can supply the following command-line switches to the Artix executable:

`-BUSconfig_domains_dir` *DomainDir*	Look for the Artix configuration file in the directory, *DomainDir*.
`-BUSdomain_name` *DomainName*	The name of the Artix configuration file is *DomainName*`.cfg`.
`-BUSname` *ConfigScope*	Initialize the Artix Bus instance with the settings from the *ConfigScope* configuration scope in the *DomainName*`.cfg` configuration file

These command-line switches can be supplied to the Artix container executable, `it_container`, or any standalone Artix executable (assuming the `main()` function was implemented to pass command-line arguments to the `IT_Bus::init()` function).

Running the application with configuration switches

Using the preceding configuration command-line switches, you can customize the configuration for the Hello World service plug-in and client.

For example, to run the Hello World application with a customized configuration, do the following:

1. Copy the sample configuration from Example 6 on page 15 into the text file, *ArtixExampleDir*`/etc/hello_world.cfg`,
2. Open a command prompt, change directory to *ArtixExampleDir*`/server`, and enter the following command:

```
it_container -BUSname artix_example.server
   -BUSconfig_domains_dir ../etc
   -BUSdomain_name hello_world
   -publish -deploy deployHelloWorldSOAPService.xml
```

3. Open another command prompt, change directory to *ArtixExampleDir*`/client`, and enter the following command:

```
GreeterClient -BUSname artix_example.client
   -BUSconfig_domains_dir ../etc
   -BUSdomain_name hello_world
```

Environment variables for configuration

Instead of supplying the `-BUSconfig_domains_dir` and the `-BUSdomain_name` switches at the command line, you can specify the Artix configuration file location using the following environment variables:

`IT_CONFIG_DOMAINS_DIR`	Environment variable that specifies the directory in which the Artix configuration file is located.
`IT_DOMAIN_NAME`	Environment variable that specifies the domain name, *DomainName*, from which the name of the Artix configuration file, *DomainName*`.cfg`, is derived.

There is no environment variable corresponding to the `-BUSname` command-line switch. Hence, the `-BUSname` command-line switch still needs to be supplied to the command, even if the preceding environment variables are set.

See ***Configuring and Deploying Artix Solutions*** for more details on environment variables.

Server Programming

This chapter describes how to develop an Artix server, which can be based either on the container model or on the standalone model. In many cases, the bulk of the server code can be generated by the Artix WSDL-to-C++ compiler, leaving the programmer to implement just the servant classes.

Programming with the Container Model

The Artix container model is a way of building and deploying Artix servers, which is based on the idea that an Artix server can be divided into two pieces: a container piece and a *service plug-in* (or plug-ins). The container piece is a standard executable, `it_container`, which is the same for all Artix servers. The service plug-in is a shared library or DLL, which must be implemented by an Artix server programmer.

This section provides a general overview of the container architecture and how it affects server-side programming. In this model, the programmer can focus on implementing service plug-ins instead of implementing standalone server executables.

Container Architecture

Figure 2 shows an overview of the Artix container architecture, which shows how a service plug-in fits into the container model. The server programmer is responsible for implementing a service plug-in, which is deployed by loading it into the Artix container.

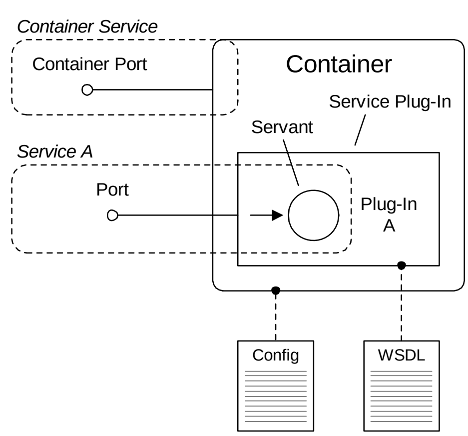

Figure 2: *Architecture of the Artix Container*

The basic elements of the Artix container architecture are:

- Container.
- Artix configuration file.
- Service plug-in.
- Servant.
- WSDL contract.

Container

The Artix container provides a convenient model for deploying Artix services, removing the need for much of the boilerplate code that would otherwise appear in the `main()` function of a traditional stand-alone server. As shown in Figure 2, a WSDL service deployed using the container model, consists of the following major components:

- *Container executable*—the container is an executable, `it_container`, capable of loading service plug-ins.
- *Service plug-ins*—plug-ins are packaged either as shared libraries or DLLs, depending on the platform. The plug-ins are loaded into the container using the dynamic linking capabilities of the operating system.

An added benefit of deploying services in a container is that the container supports elementary operations for administering services, as follows:

- Deploy new services to the container.
- List all services in the container.
- Stop a specified service.
- Start a specified service.
- Publish a URL, a reference, or a WSDL contract for a specified service.

These operations are supported by a dedicated WSDL port which provides access to the *container service*. To administer the container, Artix provides a command-line utility, `it_container_admin`. For details, see *Configuring and Deploying Artix Solutions*.

Artix configuration file

The Artix configuration file provides general-purpose configuration data for the container process (see "Adding Configuration to the Application" on page 14 for details on configuration). You can specify which configuration scope applies to the container by passing the `-BUSname` command-line switch when you launch the container, where the argument to the `-BUSname` switch is the *Bus ID*.

> **Note:** For each container process, it is possible to specify a single Bus ID and only *one* Bus instance is created. That is, service plug-ins that load into a container cannot be configured independently. In view of this limitation, only related service plug-ins should be loaded into the same container instance. The Artix container is *not* an application server.

Service plug-in

A service plug-in is a component that contains the implementation of one or more WSDL services. It consists of the following:

- *Shared library or DLL*—a dynamically loadable library that contains the code for the service plug-in.
- *Shared library dependencies file*—a dependencies file that lists the Artix plug-ins on which this plug-in depends (can be empty).
- *Deployment descriptor file*—an XML file that is passed to the Artix container in order to deploy the service plug-in.
- *WSDL contract (or contracts)*—the contract for the WSDL services provided by the plug-in.

Servant

A servant is a C++ class that implements operations from a WSDL port type (or, sometimes, from multiple port types).

It is important to understand that a servant is *not* identical to a service. The separation of the implementation from the service permits greater flexibility in the way services are implemented. For example, in some cases a service is implemented by multiple servants; in other cases, multiple services are implemented by a single servant.

A servant is not associated with a service until it is registered. See “Registering Static Servants” and “Registering Transient Servants”.

WSDL contract

A service plug-in is always associated with a WSDL contract (in some cases, with multiple WSDL contracts). The WSDL contract describes the interfaces (WSDL port types) for all of the services deployed in the plug-in.

The WSDL contract must be made available to the container through one of the mechanisms described in “How Services Locate WSDL Contracts”.

Multiple Services in a Container

Consider the case where you have two services, service A and service B, that you want to deploy into the same container. Figure 3 shows two alternative approaches to deploying these services. In the first approach (Figure 3 (a)), each service is deployed separately in its own plug-in. In the second approach (Figure 3 (b)), the services are deployed together in a single plug-in. Generally, if the services are closely related, it makes sense to deploy them in a single plug-in (as shown in Figure 3 (b)). Deploying the services as a single plug-in makes it easier for the two services to interact with each other and to share common data.

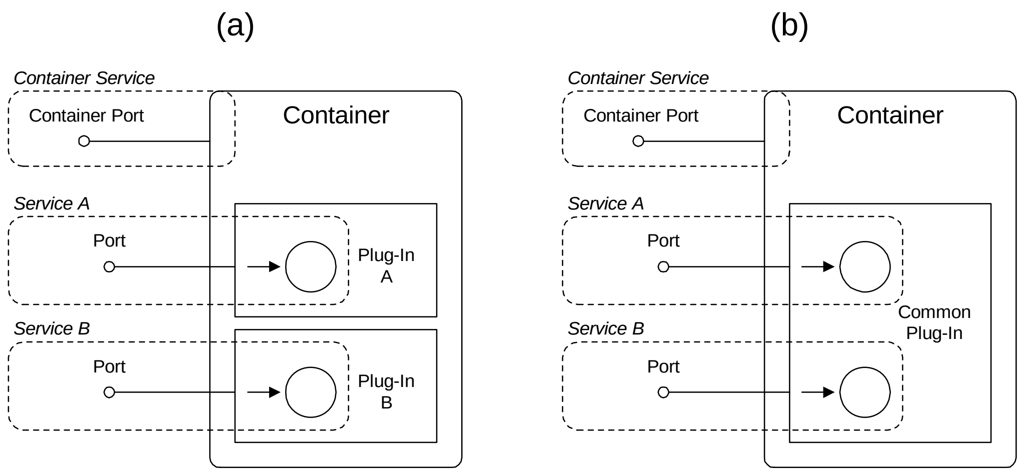

Figure 3: *Multiple Services in Separate (a) or Common (b) Plug-In*

Separate plug-ins for each service

Generating separate plug-ins for each service is the default model of deployment, which you get if you use `wsdltocpp` to generate the service plug-in.

Example 7 shows the implementation of the `bus_init()` function in a service plug-in, `Service_A_PlugIn`, that registers just a single service, *Service A*. The `bus_init()` function for the other service, *Service B*, is implemented in a similar way in a separate plug-in class, `Service_B_PlugIn`.

Example 7: *One Service Registered in each Plug-In*

```
// C++
void
Service_A_PlugIn::bus_init(
) IT_THROW_DECL((Exception))
{

    WSDLService* wsdl_service =
        get_bus()->get_service_contract(m_service_A_qname);

    get_bus()->register_servant(
        m_servant_A,
        *wsdl_service_A
    );
}
```

Common plug-in for all services

Typically a more efficient solution, if you want to deploy a number of closely related services, is to combine the different services in a single service plug-in.

Example 8 shows the implementation of the `bus_init()` function for a common plug-in, which combines the registration of both Service A and Service B.

Example 8: *Multiple Services Registered in a Plug-In*

```
// C++
void
CommonPlugIn::bus_init(
) IT_THROW_DECL((Exception))
{
    WSDLService* wsdl_service_A =
        get_bus()->get_service_contract(m_service_A_qname);

    get_bus()->register_servant(
        m_servant_A,
        *wsdl_service_A
    );

    WSDLService* wsdl_service_B =
        get_bus()->get_service_contract(m_service_B_qname);

    get_bus()->register_servant(
        m_servant_B,
        *wsdl_service_B
    );
}
```

Service with Multiple Ports

Consider the case where a single service, service A, exposes two different WSDL ports. For example, one of the ports might accept only insecure connections while the other port accepts only secure connections.

Figure 4 shows two different approaches to activating the ports. In the first approach (Figure 4 (a)), a single servant object is registered against both ports, so that request messages from both ports are directed to the same servant object. In the second approach (Figure 4 (b)), each port is registered against a different servant object. The second approach (servant for each port) is useful in cases where you need to fine-tune the servant implementation for each of the WSDL ports. For example, if one of the ports is insecure, you might want to implement a corresponding servant object that restricts access to sensitive resources.

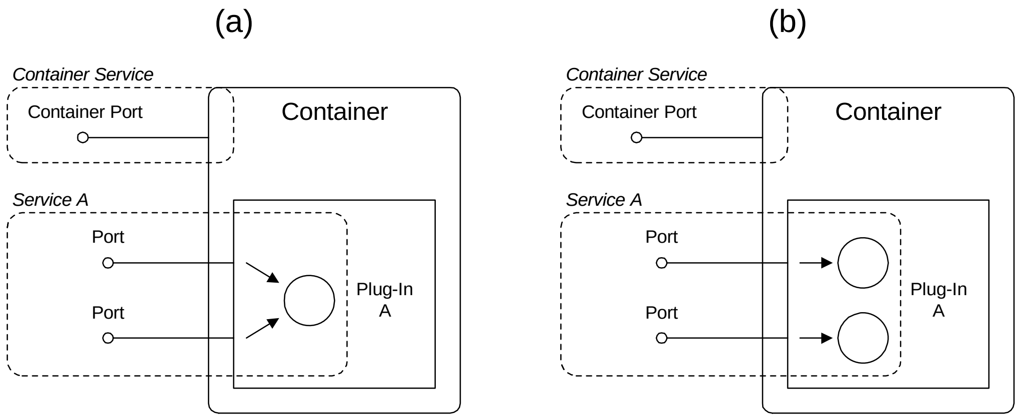

Figure 4: *Multi-Port Service Registered against a Single Servant (a), or Multiple Servants (b)*

Activating all ports together

If you activate a service's ports together, you associate all of the ports with a single servant object. For details of how to program this approach, see "Activate all ports together" on page 54.

Activating ports individually

If you activate a service's ports individually, you can optionally associate each of the WSDL ports with a different servant object. For details of how to program this approach, see "Activate ports individually" on page 55.

Implementing a Servant Class

The main task required of an Artix server programmer is the implementation of one or more servant classes. A *servant class* provides the implementation of a WSDL service. Because the servant member functions are generated from a particular WSDL port type, a given servant class can implement only WSDL services that have the same WSDL port type.

Figure 5 shows the class hierarchy for a typical servant implementation class, *PortType*`Impl`.

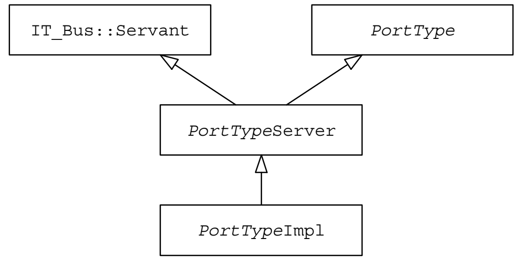

Figure 5: *Class Hierarchy for the Servant Implementation Class*

The following classes appear in this hierarchy:

- `IT_Bus::Servant` class—is the base class for all servant types. It declares a few standard member functions.
- *PortType* class—an abstract class generated from the WSDL port type named *PortType*. This class contains a function corresponding to each of the WSDL operations in the *PortType* port type.
- *PortType*`Server` class—the server skeleton class, which is generated by the `wsdltocpp` utility when the `-server` switch is supplied. The skeleton class includes code for dispatching the operations in the *PortType* port type.
- *PortType*`Impl` class—the servant class, which provides the implementation of the *PortType* port type.

  You must implement this class in order to implement a WSDL service.

Generating the servant class

To generate an outline implementation of the servant class, invoke the `wsdltocpp` command as follows:

```
wsdltocpp -i port_type
   -e web_service_name
   -server
   -impl
   -m [NMAKE|UNIX]:library
   -plugin[:plugin_name]
   -deployable
   WSDLContractFile
```

In this example, the last item on the command line, *WSDLContractFile*, is the path name (or possibly URL) of the WSDL contract. The switches shown in the preceding command have the following meaning:

`-i` *port_type*	Specifies the name of the port type for which the tool will generate code.
`-e` *web_service_name* [:*port_list*]	Specifies the name of the service for which the tool will generate code.

`-server`	Generates stub code for a server (cannot be combined with the `-client` switch).
`-impl`	Generates an outline implementation of the servant class.
`-m {NMAKE | UNIX}:[executable | library]`	Used in combination with `-impl` to generate a makefile for the specified platform (`NMAKE` for Windows or `UNIX` for UNIX). You can specify that the generated makefile builds an executable, by appending `:executable`, or a library, by appending `:library`. For example, the options, `-impl -m NMAKE:executable`, would generate a Windows makefile to build an executable.
`-plugin [:plugin_name]`	Generates a service plug-in. You can optionally specify the plug-in name by appending `:plugin_name` to this option. If no plug-in name is specified, the default name is *<ServiceName><PortTypeName>*. The service name, *<ServiceName>*, is specified by the `-e` option.
`-deployable`	(Used with `-plugin`.) Generates a deployment descriptor file, deploy*<ServiceName>*.xml, which is needed to deploy a plug-in into the Artix container.

Implementing the constructor

You can implement any kind of constructor you like for the servant implementation class. There is, however, one condition that must always be fulfilled: one of the arguments to the *PortType*`Impl()` constructor must be of type `IT_Bus::Bus_ptr` and the `bus` argument must be passed into the base constructor, *PortType*`Server()`.

For example, you can implement a simple constructor for the `Bank` port type, as follows:

```
// C++
BankImpl::BankImpl(IT_Bus::Bus_ptr bus) : BankServer(bus)
{
    ...
}
```

Implementing WSDL operations

For every operation belonging to a particular port type in the WSDL contract, the `wsdltocpp` compiler generates a corresponding member function in the servant class. The C++ function signatures are derived from the WSDL operation definitions, as follows:

- First come the parameters corresponding to the input messages,
- Next come the parameters corresponding to the input/output messages (messages sent both to and from a service),

- And finally come the parameters corresponding to the output messages.

None of the messages are represented as a return value in C++. Hence, C++ functions corresponding to WSDL operations *always* return the `void` type. For more details about mapping WSDL operations to C++ functions, see "Operations and Parameters" on page 81.

For example, the `create_account` operation in the `Bank` port type maps to the following C++ member function:

```
// C++
void
BankImpl::create_account(
    const IT_Bus::String &account_name,
    WS_Addressing::EndpointReferenceType &_return
) IT_THROW_DECL((IT_Bus::Exception))
{
    ...
}
```

The `account_name` string parameter corresponds to an input message and the `_return` parameter, of `WS_Addressing::EndpointReferenceType` type, corresponds to an output message. The `WS_Addressing::EndpointReferenceType` type enables a reference to a WSDL service to be transmitted over the wire. A reference encapsulates the location information for a particular WSDL service. For more details about references, see "Endpoint References" on page 119.

Implementing runtime callbacks

There are some standard functions that the servant class inherits from `IT_Bus::Servant`. You can optionally override these functions to receive callback notifications from the Artix runtime when certain events occur. The following callback functions are inherited from `IT_Bus::Servant`:

```
// C++
// Servant functions inherited from IT_Bus::Servant.
void activated(IT_Bus::Port& port);

void deactivated(IT_Bus::Port& port);

IT_Bus::Servant* clone() const;
```

Whenever a WSDL port is activated or deactivated, Artix calls `activated()` or `deactivated()`, respectively, to notify the servant of this event. If you do not implement these functions, the server skeleton code provides default implementations, which do nothing. These functions are typically only needed by advanced applications.

The `clone()` function gets called by the Artix runtime to create a new servant instance. An implementation of the `clone()` function is required to support certain threading policies on the server side. For more details see "Servant Threading Models" on page 103.

Calling Bus APIs

The servant application code can also access a variety of Artix APIs through the Bus object. The Bus object can be conveniently accessed by calling the `get_bus()` member function, which is implemented by the `IT_Bus::Servant` base class:

```
// C++
virtual Bus_ptr get_bus() const;
```

One of the most common reasons for accessing the Bus instance, is in order to write to or read from an *Artix context*. Artix contexts provide a mechanism for accessing data in message headers or for fine-tuning Artix behavior by setting policies programmatically. For more information about Artix contexts, see “Artix Contexts” on page 153.

Implementing the Plug-In Class

The service plug-in class provides the entry point for initializing and shutting down the plug-in. For very simple applications, you can use the default, generated implementation of the plug-in class. For most applications, however, you will probably need to make some modifications to the plug-in class.

Plug-in functions

The service plug-in class essentially provides a programmer with two hooks:

- `bus_init()`—a function called as the plug-in initializes.
- `bus_shutdown()`—a function called as the plug-in shuts down.

The primary purpose of the `bus_init()` function is to let you register servant objects. By registering a servant object, you create an association between the servant object and a particular WSDL service, such that requests received by the WSDL service are invoked on the servant object. If you are using service activators, however, you would typically delegate servant registration to the service activators.

The `bus_shutdown()` function enables you to perform clean-up tasks as the Bus and the plug-in are shutting down.

Summary of container programming

The following points summarize how to program an Artix server in the container programming model:

- The `bus_init()` and `bus_shutdown()` functions in the plug-in class take the place of a `main()` function.
- The plug-in class is primarily used for registering service activators and for registering and deregistering servants (in `bus_init()` and `bus_shutdown()` respectively).

- There is no need to call either the `IT_Bus::init()` function or the `IT_Bus::Bus::shutdown()` function. The container looks after initializing and shutting down the Bus object.
- Call `get_bus()` to get the `IT_Bus::Bus` instance.
- Instead of hard-coding the location of a WSDL contract, you can find a contract using the `IT_Bus::Bus::get_service_contract()` function.

Generating the plug-in class

To generate a default implementation of the service plug-in class, invoke the `wsdltocpp` command as follows:

```
wsdltocpp -i port_type
   -e web_service_name
   -server
   -impl
   -m [NMAKE|UNIX]:library
   -plugin[:plugin_name]
   -deployable
   WSDLContractFile
```

In this example, the last item on the command line, *WSDLContractFile*, is the path name (or possibly URL) of the WSDL contract. The switches shown in the preceding command are explained in "Generating the servant class" on page 25.

The `wsdltocpp` utility with the `-plugin` switch generates the following files containing a default implementation of the service plug-in class:

<*web_service_name*><*port_type*>`PlugIn.h`
<*web_service_name*><*port_type*>`PlugIn.cxx`

Where <*web_service_name*> is the WSDL service specified by the `-e` switch of the `wsdltocpp` command and <*port_type*> is the port type specified by the `-i` switch.

Plug-in constructor

The plug-in constructor is called as the plug-in is loaded. This is a convenient place to create basic objects that the plug-in needs.

Example 9 shows an example of a constructor for the `BankService` plug-in. This constructor creates a service activator instance, `m_service_activator`, that is responsible for activating the `BankService` service and a QName instance, `m_service_qname`, that holds the name of the `BankService` service.

Example 9: *Sample Plug-In Constructor for the Bank Service Plug-In*

```
// C++
BankServantBusPlugIn::BankServantBusPlugIn(
    Bus_ptr bus
) IT_THROW_DECL((Exception))
  :
    BusPlugIn(bus),
    m_service_activator(0),
```

Example 9: *Sample Plug-In Constructor for the Bank Service Plug-In*

```
    m_service_qname("", "BankService",
   "http://www.iona.com/bus/demos/bank")
{
    // complete
}
```

bus_init() function

The `bus_init()` function is called either during Bus initialization or just after the plug-in is loaded. The `bus_init()` function is the place to put the code that registers servants with the Bus. If the plug-in uses service activators, the `bus_init()` function should register the service activators with the Bus and then delegate servant registration to the service activators.

Example 10 shows an implementation of `bus_init()` that registers a service activator object against the `BankService` service. The code then explicitly calls `activate_service()` on the service activator instance, which has the effect of registering a Bank servant with the Bus

Example 10: *Sample Implementation of bus_init()*

```
// C++
void
GreeterServantBusPlugIn::bus_init(
) IT_THROW_DECL((Exception))
{
    try
    {
       m_service_activator
        =  new IT_Bus_Services::ServiceActivatorImpl(get_bus());

        if (0 == m_service_activator.get())
        {
         String error("Failed to initialize ServiceActivator");
          error += " for service, ";
          error += m_service_qname.to_string();
          throw Exception(error);
        }

        ServiceActivator::register_sa(
            get_bus(),
            m_service_qname,
            m_service_activator.get()
        );

        m_service_activator->activate_service(m_service_qname);
    }
    catch (const IT_Bus::Exception & ex)
    {
        throw Exception(ex);
    }
}
```

bus_shutdown() function

The bus_shutdown() function is called when the Bus instance is shut down (that is, when the container calls IT_Bus::Bus::shutdown()).

Example 11 shows an implementation of bus_shutdown() that deactivates the BankService service, which results in de-registration of the Bank servant.

Example 11: *Sample Implementation of bus_shutdown()*

```
// C++
void
GreeterServantBusPlugIn::bus_shutdown(
) IT_THROW_DECL((Exception))
{
    m_service_activator->deactivate_service(
        m_service_qname
    );
}
```

Implementing the Service Activator Class

The service activator class provides the entry point for creating, registering and deregistering servants. In general, this class is used to manage the lifecycle of an Artix service. If the relevant member functions of the service activator class are properly implemented, it should be possible to deactivate and then re-activate a service without needing to shut down the entire service plug-in.

Service activator functions

The service plug-in class provides two functions that control the lifecycle of an Artix service, as follows:

- activate_service()—a function called either from within bus_init() or whenever the it_container_admin -deploy command is executed.

  The purpose of the activate_service() function is to perform all of the housekeeping tasks necessary to start up an Artix service, including the creation of a servant object and the registration of that servant object with the Bus.
- deactivate_service()—a function called either from within bus_shutdown() or whenever the it_container_admin -removeservice command is executed.

  The purpose of the deactivate_service() function is to perform all of the housekeeping tasks necessary to shut down an Artix service, including deregistration of the service and deletion of the associated servant object.

Related container administration commands

The lifecycle functions provided by the service activator class are closely related to the following it_container_admin commands:

- it_container_admin -deploy—the effect of issuing this command depends on whether this is the first or subsequent deployment, as follows:
    - *First deployment*—load and initialize the service plug-in. The container calls bus_init(), which is normally programmed to call activate_service() for each of the WSDL services.
    - *Subsequent deployment (re-deploy)*—activate any inactive services. The container calls activate_service() on each of the registered service activators, but only if the service is currently inactive. The container does not call bus_init() in this case.

  Note: Artix does not currently provide an administration command that re-activates a single service at a time. The -deploy command re-activates all of the inactive services from the specified plug-in.

- it_container_admin -removeservice—de-activate a specific service. When you issue the -removeservice command, the container calls deactivate_service(), but only if the specified service is currently active.

For more details about the it_container_admin command-line utility, see ***Configuring and Deploying Artix Solutions***.

Generating the service activator class

The service activator class is generated by the wsdltocpp command at the same time as the plug-in class. For details of how to generate a default implementation of the service activator class and the plug-in class, see "Generating the plug-in class" on page 29.

The wsdltocpp utility generates the following files containing a default implementation of the service activator class:

<*port_type*>_service_activator_impl.h
<*port_type*>_service_activator_impl.cxx

Where <*port_type*> is the port type specified to wsdltocpp by the -i switch.

activate_service() function

The activate_service() function is called either from the body of the bus_init() function or whenever the it_container_admin -deploy command is issued. The activate_service() function is the appropriate place to put the code that creates and registers servants.

Example 12 shows an implementation of `activate_service()` that registers a Bank servant, thereby associating it with the `BankService` WSDL service.

Example 12: *Sample Implementation of activate_service()*

```
// C++
void
ServiceActivatorImpl::activate_service(
    const IT_Bus::QName& service_name
) IT_THROW_DECL((IT_Bus::Exception))
{
    if (m_impl==0) {
        m_impl = new COM_IONA_BANK::BankImpl(
            m_bus.get()
        );
    }

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

    if (wsdl_service != 0)
    {
        m_bus->register_servant(
            *m_impl,
            *wsdl_service
        );
    }
}
```

In this example, it is assumed that the service activator instance was registered as shown in Example 10 on page 30—that is, the service activator instance is registered *only* against the `BankService` service. Hence, it follows that the `activate_service()` function shown in Example 12 will only be called when `service_name` equals the `BankService` QName.

Advanced applications might choose to register a service activator instance against several different services. In that case, you would need to examine the service QName, `service_name`, in order to decide which servant to activate.

deactivate_service() function

The `deactivate_service()` function is called either from the body of the `bus_shutdown()` function or whenever the `it_container_admin -removeservice` command is issued.

Example 13 shows an implementation of `deactivate_service()` that deregisters and deletes the Bank servant that was registered by `activate_service()`.

Example 13: *Sample Implementation of deactivate_service()*

```
// C++
void
ServiceActivatorImpl::deactivate_service(
    const IT_Bus::QName& service_name
)
{
    m_bus->remove_service(service_name);

    delete m_impl;
    m_impl = 0;
}
```

Programming with the Standalone Model

If you prefer not to deploy your Artix server using the container model, you can opt for the standalone model instead. In the standalone model, you are responsible for writing the server's `main()` function directly. Instead of building a plug-in, the servant code and `main()` function are linked together and built as a standalone executable.

The standalone model is simpler than the container model in some respects, but it has the disadvantage that you cannot monitor a standalone executable using the Artix management console.

Generating the standalone server

To generate an outline implementation of a standalone server, invoke the `wsdltocpp` command as follows:

```
wsdltocpp -i port_type
   -e web_service_name
   -sample
   -impl
   -m [NMAKE|UNIX]:executable
   WSDLContractFile
```

In this example, the last item on the command line, *WSDLContractFile*, is the path name (or possibly URL) of the WSDL contract.

The switches shown in the preceding command have the following meaning:

`-i` *port_type* — Specifies the name of the port type for which the tool will generate code.

`-e` *web_service_name* [:*port_list*] — Specifies the name of the service for which the tool will generate code.

`-sample` — Generates code for a server main function and a client main function.

`-impl` — Generates an outline implementation of the servant class.

`-m {NMAKE | UNIX}:[executable | library]` — Used in combination with `-impl` to generate a makefile for the specified platform (`NMAKE` for Windows or `UNIX` for UNIX). You can specify that the generated makefile builds an executable, by appending `:executable`, or a library, by appending `:library`. For example, the options, `-impl -m NMAKE:executable`, would generate a Windows makefile to build an executable.

Sample main() function

Example 14 shows the basic outline of a server `main()` function. In this example, the `main()` function registers a single `GreeterImpl` servant against the `HelloWorldSOAPService` service.

Example 14: *Sample main() Function for Standalone Server*

```
// C++
#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_bus/fault_exception.h>
#include <it_cal/iostream.h>

IT_USING_NAMESPACE_STD

#include "GreeterImpl.h"

using namespace COM_IONA_HELLO_WORLD_SOAP_HTTP;
using namespace IT_Bus;

int main(int argc, char* argv[])
{
    cout << " Greeter service" << endl;

    try
    {
1           IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);

2           GreeterImpl servant(bus);

            IT_Bus::QName service_name_0("",
   "HelloWorldSOAPService",
   "http://www.iona.com/hello_world_soap_http");
```

Example 14: *Sample main() Function for Standalone Server*

```
        bus->register_servant(
                            servant,
                            "../etc/hello_world.wsdl",
                            service_name_0
                            );

3       bus->run();
    }
    catch(IT_Bus::Exception& e)
    {
        cout << "Error occurred: " << e.message() << endl;
        return -1;
    }
    return 0;
}
```

The preceding code example can be explained as follows:

1. When writing the server `main()` function, you need to initialize the Artix Bus explicitly by calling the `IT_Bus::init()` function.

   It is important also to pass the command line arguments to the `IT_Bus::init()` function, otherwise the server would not respond to the standard Artix command-line options.
2. This example creates a single servant object, of `GreeterImpl` type, and registers this servant against the `HelloWorldSOAPService` service. Artix supports many different options for registering servant options—for more details, see “Registering Static Servants” on page 52 and “Registering Transient Servants” on page 56.
3. Call `IT_Bus::Bus::run()` to send the main thread to sleep. This allows the background threads to continue processing incoming request messages.

Default Servants

A default servant enables you to implement a scalable factory pattern, enabling you to replace multiple servants of the same type by a single servant.

Introduction to Default Servants

A *default servant* enables you to implement multiple services of the same type, using only a *single* servant instance. In many respects, the default servant programming model is similar to the transient servant programming model (for example, see “Transient Servants” on page 47), except that multiple servant instances are now replaced by a single default servant instance. The advantage of the default servant model is its smaller footprint, in terms of memory and other resources.

Figure 6 shows an example of how a default servant could be used in a bank application. The `Bank` service creates and provides access to an unlimited number of account instances. Each account is accessed through a unique service (for example, `john.doe`). These `Account` services are created dynamically.

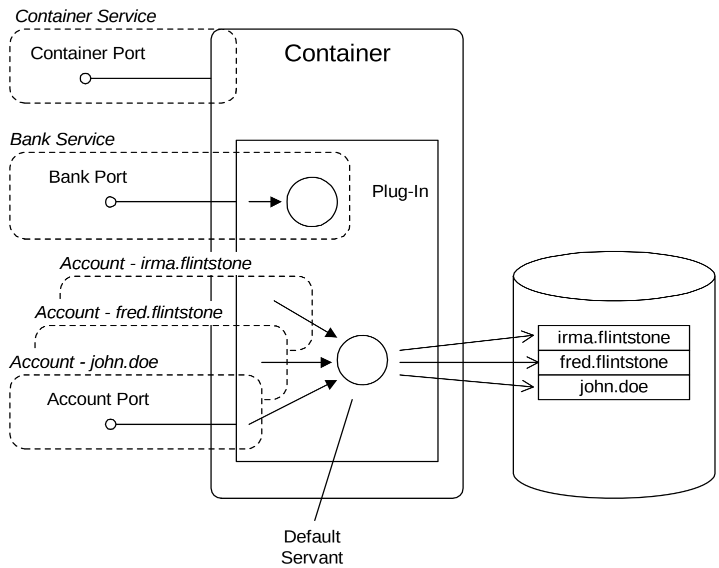

Figure 6: *Default Servant Implementing Multiple Account Services*

Factory pattern

A default servant is typically deployed in the context of a factory pattern. For example, Figure 6 shows a Bank service, which plays the role of a factory object, and a collection of cloned Account services, which are created and managed by the Bank service.

The role played by each of the servants, for Bank and Account services, can be described as follows:

- *Bank servant*—the Bank servant is responsible for creating and finding Account service instances. Because the accounts are implemented using a default servant, the bank does not need to create and register individual servants for every new account. Instead, the bank creates an account as follows:

  i. Create a record to hold the account details (for example, by creating a database record).

  ii. Generate a unique endpoint reference for the account service instance, based on a unique service ID.

  In effect, each new service has a unique identity and an associated data record, but a new servant is not created for the service.

- *Default servant for accounts*—a single default servant instance processes incoming requests for all of the account services. Hence, during an operation invocation, the default servant needs to have some way of finding out the identify of the account service for which it is acting. The current *service ID* can be obtained from the *address context*—see "The Server Address Context" on page 40 for details.

Service ID

A *service ID* is a unique identifier for a cloned service. For example, in Figure 6, the account names, `john.doe`, `fred.flintstone`, and `irma.flintstone` are service IDs.

Template service

To give you the ability to define an unlimited number of WSDL services, Artix lets you define a *template service* in the WSDL contract. A template service is defined using the same syntax as a regular service. The only additional condition that a template service must obey is that the endpoint address should conform to a *placeholder* format (for details, see "SOAP template service" on page 57 and "CORBA template service" on page 58).

For example, the following WSDL fragment shows a template service for accounts services. In this case, the placeholder format for the HTTP address is `http://localhost:0`.

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
        ... >
    ...
    <service name="AccountService">
            <port name="AccountPort" binding="tns:AccountBinding">
                <soap:address location="http://localhost:0" />
                <http-conf:server HonorKeepAlive="false"/>
                <http-conf:client Connection="close"/>
            </port>
    </service>
</definitions>
```

Cloned services

Whenever you generate a new reference using the default servant programming model, you are implicitly creating a *cloned service* based on a template service. This is similar to the concept of a cloned service in the context of transient servants—see "Transient Servants" on page 47.

For a default servant, you can create a cloned service by calling the `IT_Bus::Service::get_endpoint_reference_with_id()` function—see "Service functions" on page 40.

Supported transports

Default servants are supported by the following transports:

- *SOAP/HTTP*,
- *CORBA/IIOP*,
- *Tunnel*.

Functions Defined on IT_Bus::Service

Generally, in order to activate a service in Artix, you need to obtain a service object, of `IT_Bus::Service` type, and register one or more servant objects with this service.

For the default servant programming model, you need two functions that you can call on the `IT_Bus::Service` class, as follows:

- A function to register the default servant with the template service and,
- A function to clone new services from the template service.

These functions are, in fact, defined on the `IT_Bus::ServerService` class, which is an alias of `IT_Bus::Service`.

ServerService class

The `IT_Bus::ServerService` class, which is an alias of `IT_Bus::Service`, provides functions to support the default servant programming model, as shown in Example 15.

Example 15: *Some Member Functions in IT_Bus::ServerService*

```
// C++
namespace IT_Bus {
    ...
    class IT_BUS_API ServerService : public ServiceBase
    {
      public:

        virtual void
        register_default_servant(
            Servant &       servant,
            const String &  port_to_register = IT_BUS_ALL_PORTS
        ) = 0;
        ...
        virtual WS_Addressing::EndpointReferenceType
        get_endpoint_reference_with_id(
            const String & instance_id,
            const String & port_to_register = IT_BUS_ALL_PORTS
        ) = 0;
        ...
    };
    ...
};
```

Service functions

The member functions shown in Example 15 can be explained, as follows:

- `register_default_servant()`—activates the given service and associates the default servant, `servant`, with the service. If you use the second argument, `port_to_register`, to specify a particular port, only that port will be activated; otherwise, all of the service's ports are activated.
- `get_endpoint_reference_with_id()`—returns an endpoint reference to a newly-cloned service, which is identified by the given service ID, `instance_id`. The significance of the ID depends on the transport, as follows:
    - *SOAP/HTTP*—the URL address of the cloned service is obtained by appending the ID, *ReferenceID*, to the end of the template service's URL.

      For example, if the template service's URL is `http://enghost:2048/Account`, the cloned service's URL would be `http://enghost:2048/Account/`*ReferenceID*.
    - *IIOP*—the ID is used as the CORBA Object ID, which is ultimately embedded in a CORBA Interoperable Object Reference (IOR). The IOR is then stored inside the endpoint reference.
    - *Tunnel*—similarly to the IIOP transport, the tunnel transport uses the ID as the CORBA Object ID.

Note: The `ServerService` class (and the `IT_Bus::Service` class, which is an alias of it) also supports a function, `get_reference_with_id()`, that returns a legacy reference type, `IT_Bus::Reference`. This function is provided solely for backward compatibility reasons.

The Server Address Context

In contrast to a regular servant, which implements a unique service instance, a default servant implements an *unlimited* number of service instances. In the course of an invocation, therefore, a default servant needs some way of finding out which service it represents.

The mechanism that enables default servants to discover the current service identity is by obtaining the value of the *server address context*. The address context is a data type that can be retrieved during an invocation using the Artix context mechanism.

AddressContext class

Example 16 shows the `IT_Bus::AddressContext` class, whose instances can be accessed from within a server invocation.

Example 16: *The IT_Bus::AddressContext Class*

```
// C++
namespace IT_Bus
{
    ...
    class IT_CONTEXT_ATTRIBUTE_API AddressContext
      : public Context
    {
      public:
        ...
        virtual const IT_Bus::String&
        get_context() const;
        ...
        virtual const IT_Bus::String&
        get_full_address() const;

      protected:
        ...
    };
}
```

AddressContext functions

The `AddressContext` class in Example 16 provides the following functions for accessing the address context data:

- `get_context()` function—obtain an ID string that identifies the current cloned service. The ID string returned from this function is the same as the ID string that is passed to the `IT_Bus::Service::get_endpoint_reference_with_id()` function—see "Functions Defined on IT_Bus::Service" on page 39.
- `get_full_address()` function—obtain the full address of the current cloned service. The return value from this function depends on the transport, as follows:
    - *SOAP/HTTP*—returns the URL address for the current cloned service. For example, if the current service has an ID of *ReferenceID*, a typical return value would be:

      `http://enghost:2048/Account/`*`ReferenceID`*
    - *IIOP*—returns the full IOR (with embedded Object ID) for the current cloned service.
    - *Tunnel*—same as IIOP.

Obtaining an AddressContext instance

An `AddressContext` instance can be obtained using the Artix context API, but it is only available during an operation invocation—that is, during an upcall on the servant function that results from an incoming invocation request.

To obtain the address context data, first get a pointer to a request context container (of `IT_Bus::ContextContainer` type) and then call `get_context_data()`, passing in the string constant, `IT_ContextAttributes::SERVER_ADDRESS_CONTEXT`.

For more details on Artix contexts, see “Artix Contexts” on page 153.

Implementing a Factory

When using a default servant to implement a collection of Account services, the associated factory service, of type `Bank`, plays a crucial role. The Bank member functions that are responsible for creating and finding account objects must be written to fit the default servant programming model. In particular, you must call a special function, `IT_Bus::Service::get_endpoint_reference_with_id()` in order to create each instance of a cloned Account service.

Bank factory implementation

Example 17 shows a sample implementation of the `BankImpl` servant class, where the managed Account objects are implemented using a default servant. The implementation of the constructor and two member functions, `create_account()` and `get_account()`, are shown here.

Example 17: *Bank Factory that Uses a Default Servant for Accounts*

```
   // C++
   #include "BankImpl.h"
   #include <it_cal/cal.h>
   #include <it_cal/iostream.h>

   using namespace IT_Bank;
   using namespace IT_Bus;

   IT_USING_NAMESPACE_STD

1  const IT_Bus::QName ACC_SERVICE_NAME(
       "",
       "AccountService",
       "http://www.iona.com/bus/demos/bank"
   );

   BankImpl::BankImpl(IT_Bus::Bus_ptr bus) : BankServer(bus)
   {
2      IT_WSDL::WSDLService* wsdl_service =
           get_bus()->get_service_contract(ACC_SERVICE_NAME);

3      m_template_service = get_bus()->add_service(*wsdl_service);

       AccountImpl * default_servant = new AccountImpl(bus);
4      m_template_service->register_default_servant(
           default_servant
       );
   }
```

Example 17: *Bank Factory that Uses a Default Servant for Accounts*

```
void
BankImpl::create_account(
    const IT_Bus::String &account_name,
    WS_Addressing::EndpointReferenceType &_return
) IT_THROW_DECL((IT_Bus::Exception))
{
    // Check whether account already exists.
    ...
5   if ( /* Account does NOT already exist... */ )
    {
        // Create a new account for the account_name account.
6       _return =
            m_template_service->get_endpoint_reference_with_id(
                account_name
            );

7       // Create a new account record, update the database,
   etc.
        //
        ... // (not shown)
    }
    else {
        // Account already exists - throw an exception!
        ... // (not shown)
    }
}

void
BankImpl::get_account(
    const IT_Bus::String &account_name,
    IT_Bus::Reference &_return
) IT_THROW_DECL((IT_Bus::Exception))
{
    // Search for the account_name account.
    ... // (not shown)
8   if ( /* Account exists... */ )
    {
9       _return =
            m_template_service->get_endpoint_reference_with_id(
                account_name
            );

        return;
    }

    // Account not found - throw an exception!
    ... // (not shown)
}
```

The preceding code example can be explained as follows:

1. The `ACC_SERVICE_NAME` constant holds the QName of the Account template service. The template service is used as a basis for cloning Account service instances.
2. The `get_service_contract()` function locates the contract containing the specified Account service. The returned `IT_WSDL::WSDLService` object represents all of the data contained in the `service` element for the Account service.

For more details, see "How Services Locate WSDL Contracts" on page 50.

3. The `m_template_service` object, which is of `IT_Bus::Service_var` type, is a data member of the `BankImpl` class. Artix uses an `IT_Bus::Service` object to associate a service's endpoints with a particular servant (or servants).
4. Call `register_default_servant()` to associate the template service, `m_template_service`, with the default servant, of `AccountImpl` type.
5. In the body of the `BankImpl::create_account()` function, the first think you need to do is to check whether the requested account, `account_name`, already exists or not. If the account already exists, you would need to throw an exception.
6. Call `get_endpoint_reference_with_id()`, passing `account_name` as the ID, to create a new endpoint reference, of `WS_Addressing::EndpointReferenceType` type. This step effectively clones a new service from the template service. The name of the cloned service is derived by appending the specified ID (in this case, `account_name`) to the Account service URL.

   For example, if the Account service's URL is `http://enghost:2048/Account` and the account name is `john.doe`, the name of the cloned service would be `http://enghost:2048/Account/john.doe`.
7. You can use the account name as a key for creating a database record that holds the account details.
8. In the body of the `BankImpl::get_account()` function, you first need to check whether the specified account exists. If not, you would throw an exception.
9. Call the `get_endpoint_reference_with_id()` function to generate an endpoint reference with the specified ID.

Implementing a Default Servant

This section describes how to implement a default servant class for a collection of cloned Account services. A single default servant instance is sufficient to provide an implementation for all of the Account services.

The key difference between a regular servant and a default servant is that the default servant has multiple identities. Whereas a regular servant has its identity set at the time it is constructed, a default servant assumes a new identity each time it is invoked through the Artix call stack. A programmer is, therefore, obliged to discover the default servant's current identity by obtaining the *address context* for the current invocation.

Default servant class implementation

Example 18 shows a sample implementation of the Account template service, using a default servant. The implementation of the `get_balance` operation provides a typical example of how to implement a WSDL operation in a default servant.

Example 18: *Default Servant Class for Accounts*

```
// C++
#include "AccountImpl.h"
#include <it_cal/cal.h>
#include <it_cal/iostream.h>

#include <it_bus/bus.h>
#include <it_bus/service.h>
#include <it_bus_pdk/context.h>
#include <it_bus_pdk/context_attrs/context_constants.h>
#include <it_bus_pdk/context_attrs/address_context.h>

using namespace IT_Bank;

IT_USING_NAMESPACE_STD

const IT_Bus::QName AccountImpl::SERVICE_NAME("",
   "AccountService", "http://www.iona.com/bus/demos/bank");

AccountImpl::AccountImpl(
    IT_Bus::Bus_ptr bus
): AccountServer(bus)
{
}

AccountImpl::~AccountImpl()
{
}

IT_Bus::Servant*
1 AccountImpl::clone() const
{
    assert(0);
    return 0;
}

void
2 AccountImpl::get_balance(
    IT_Bus::Float & balance
) IT_THROW_DECL((IT_Bus::Exception))
{
    IT_Bus::ContextRegistry* context_registry =
        get_bus()->get_context_registry();

    IT_Bus::ContextCurrent& context_current =
        context_registry->get_current();

3   IT_Bus::ContextContainer* context_container =
        context_current.request_contexts();

4   IT_Bus::Context* result =
```

Example 18: *Default Servant Class for Accounts*

```
            context_container->get_context_data(
                IT_ContextAttributes::SERVER_ADDRESS_CONTEXT
            );

5       IT_Bus::AddressContext* address =
            dynamic_cast<IT_Bus::AddressContext*>(result);

        if (address)
        {
            // Get the account name from the address context.
6           IT_Bus::String account_name = address->get_context();

            // Consult the account_name record in the database to
            // get account balance.
7           balance = ... // (not shown)
        }
        else {
            // Could not access address context - throw an
    exception!
            ... // (not shown)
        }
}
...
```

The preceding code example can be explained as follows:

1. The `clone()` function is required for certain Artix threading policies (see “Servant Threading Models” on page 103). It is not relevant to default servants and is not used in this scenario.
2. The `get_balance()` function illustrates the basic principles of implementing an operation in a default servant. The function simply returns the account balance for a particular account. There is just one difficulty: seeing as how the default servant can represent any account instance, you have to figure out which particular account to access. To find the name of the account, you must obtain the *address context* for this invocation.
3. Obtain the context container for request contexts, `context_container`. On the server side, contexts can be used to hold miscellaneous data relevant to the current invocation. For more details about programming with contexts, see “Artix Contexts” on page 153.
4. Call `get_context_data()` on the request context container in order to obtain the address context for the current invocation. The address context is identified by the `IT_ContextAttributes::SERVER_ADDRESS_CONTEXT` string constant.
5. In order to use the address context, you must cast it first of all to the `IT_Bus::AddressContext*` type.
6. Retrieve the account name from the address context by calling `AddressContext::get_context()`. You know that the address context contains the account name, because the account name was used as the reference ID at the time the account was created (see “Implementing a Factory” on page 42).

7. You can now use the account name to retrieve the account balance from a database record.

Transient Servants

Artix allows you to generate an unlimited number of services from a single template by taking advantage of *transient servants*. This feature is useful for those cases where Artix bridges into a technology domain that maps services to object instances. Because it is usual to allow an unlimited number of objects of a particular type, it follows that this kind of bridge can work only if Artix allows an unlimited number of *services* of a particular type.

> **Note:** For highly scalable applications, it is recommended that you choose the default servant approach over the transient servant approach—see "Default Servants" on page 36.
>
> Using the transient servant approach, there is a risk that the number of transient servants could become unmanageably large. But this problem does not arise with the default servant approach, because you only need a *single* default servant to process requests for an unlimited number of services.

Figure 7 shows an example of how transient servants could be used in a bank application. The `Bank` service creates and provides access to an unlimited number of `Account` objects. Each `Account` object is accessed through a unique service (for example, `Account1`, `Account2`, and `Account3`). These `Account` services are created dynamically by registering servants as transient.

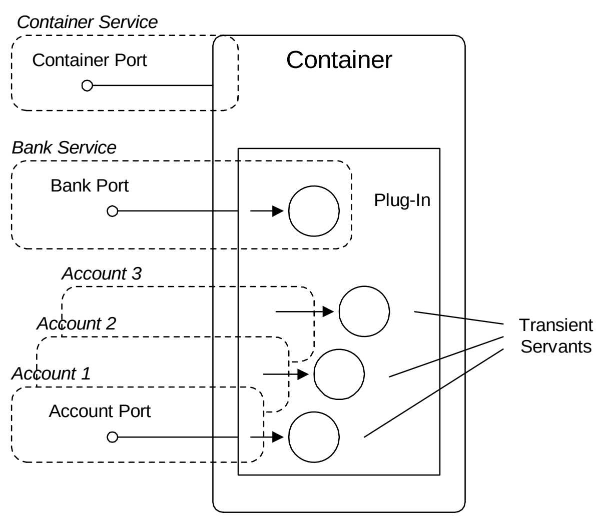

Figure 7: *Transient Servants for an Account Service*

Factory pattern

The need for transient servants commonly arises when implementing the *factory pattern*, which is a common object-oriented design pattern. At a minimum, the factory pattern involves two interfaces, as follows:

- *Creator*—an interface that provides operations for creating and finding objects of a particular type (the products). In the current example, the `Bank` port type plays the role of a creator interface.
- *Product*—an interface for the objects produced by the creator. In the current example, the `Account` port type plays the role of a product interface.

The following WSDL fragment shows the outline of a Bank port type and an Account port type, which together exemplify a factory design pattern:

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
        ... >
    ...
    <message name="create_account">
        <part name="account_name" type="xsd:string"/>
    </message>
    <message name="create_accountResponse">
        <part name="return" type="wsa:EndpointReferenceType"/>
    </message>
    ...
    <portType name="Bank">
        ...
        <operation name="create_account">
          <input name="create_account" message="tns:create_account"/>
          <output name="create_accountResponse" message="tns:create_accountResponse"/>
        </operation>
        ...
    </portType>

    <portType name="Account">
        ...
    </portType>
    ...
</definitions>
```

The `Bank` port type exposes a `create_account` operation, which creates a new account with a specified name and returns a *reference* to the newly created `Account` object. The returned reference is represented by the `wsa:EndpointReferenceType` type.

References

An endpoint reference is an XML schema type that encapsulates the information required to connect to an Artix service. Essentially, a reference contains the same information as is contained in a WSDL `service` element.

For more details about the endpoint reference type, see “Endpoint References” on page 119.

Template service

A noteworthy feature of the factory pattern is that the creator (of `Bank` type) can create an unlimited number of products (of `Account` type). Because each account instance needs to be represented by a WSDL service, this implies that Artix needs the capability to generate an unlimited number of WSDL services for the accounts. This requirement, however, is at odds with the standard approach to defining Web services, where a fixed number of WSDL services are defined explicitly in the WSDL contract.

To give you the ability to define an unlimited number of WSDL services, Artix lets you define a *template service* in the WSDL contract. A template service is defined using the same syntax as a regular service. The only additional condition that a template service must obey is that the endpoint address should conform to a *placeholder* format (for details, see "SOAP template service" on page 57 and "CORBA template service" on page 58).

For example, the following WSDL fragment shows a template service for accounts services. In this case, the placeholder format for the HTTP address is `http://localhost:0`.

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
        ... >
    ...
    <service name="AccountService">
            <port name="AccountPort" binding="tns:AccountBinding">
                <soap:address location="http://localhost:0" />
                <http-conf:server HonorKeepAlive="false"/>
                <http-conf:client Connection="close"/>
            </port>
    </service>
</definitions>
```

At runtime, Artix modifies the in-memory copy of this WSDL service by replacing the placeholder address, `http://localhost:0`, with a URL that has a specific host and port. The server then listens for operation invocations on that host and port.

Cloned services

When you register a servant object as a transient servant, Artix implicitly *clones* a new service from the template service and associates the newly cloned service with the transient servant. Artix generates a cloned service from the template service by copying the template service and then making the following changes:

- The service QName is replaced by a unique identifier (that is, unique for every cloned service).
- The placeholder address is replaced by an active endpoint address that is unique for every cloned service.

  For example, in the case of a HTTP port, the placeholder address, `http://localhost:0`, is replaced by a real IP address with a specific host and port. A unique identifier is then appended to this URL to give the address of the cloned endpoint.

How Services Locate WSDL Contracts

For all but the simplest applications, it is recommended that you do *not* hard-code the location of a WSDL contract into your service code. In place of hard-coding the contract location, Artix supports a mechanism for locating WSDL contracts based on the service QName. If you supply Artix with a service QName, Artix will then find and parse the corresponding WSDL contract.

This approach to locating WSDL contracts consists of two steps:

1. In the application code, call `IT_Bus::Bus::get_service_contract()` with a service QName argument for the WSDL service that you want to find.
2. Using the supported location mechanisms (see "Options for providing WSDL contracts" on page 51 for details), Artix searches the available WSDL contracts to find one that contains the requested WSDL service.

Example of finding a WSDL contract

Example 19 shows how to find a WSDL service element, `SOAPService`, in the namespace, `http://www.iona.com/hello_world_soap_http`, and register a servant against it, given that the Bus has access to the WSDL contract containing the service.

Example 19: *Finding a WSDL Contract Using get_service_contract()*

```
// C++
IT_Bus::QName service_qname(
  "", "SOAPService", "http://www.iona.com/hello_world_soap_http"
);

// Find the WSDL contract
IT_WSDL::WSDLService* wsdl_service = bus->get_service_contract(
    service_qname
);

// Register the servant
bus->register_servant(
    servant,
    *wsdl_service
);
```

Options for providing WSDL contracts

Artix finds WSDL contracts from the following sources, in order of priority:

1. *Contract specified on the command line*—you can provide a WSDL contract by specifying the location of the WSDL contract file on the command line. For example:

```
it_container -BUSservice_contract ../../etc/hello_world.wsdl
   -BUSname artix_example.server
   -deploy deployHelloWorldSOAPService.xml
```

2. *Contract specified in the configuration file*—you can provide a WSDL contract from the configuration file. For example:

```
# Artix Configuration File
bus:qname_alias:hello_service =
"{http://www.iona.com/hello_world_soap_http}HelloWorldSOAPService";
bus:initial_contract:url:hello_service = "../../etc/hello.wsdl";
```

   The first line of this example associates a nickname, `hello_service`, with the QName for the `HelloWorldSOAPService` service. The `bus:initial_contract:url:hello_service` variable then specifies the location of the WSDL contract containing this service.

   For more details, see *Configuring and Deploying Artix Solutions*.

3. *Contract directory specified on the command line*—you can provide a WSDL contract by specifying a contract directory on the command line. When Artix looks for a particular WSDL service, it searches all of the WSDL files in the specified directory. For example:

```
it_container -BUSservice_contract_dir ../../etc/
   -BUSname artix_example.server
   -deploy deployHelloWorldSOAPService.xml
```

   For more details, see *Configuring and Deploying Artix Solutions*.

4. *Contract directory specified in the configuration file*—you can provide WSDL contracts by specifying a list of contract directories in the configuration file. For example:

```
# Artix Configuration File
bus:initial_contract_dir = [".", "../../etc"];
```

5. *Stub WSDL shared library*—Artix can retrieve WSDL that has been embedded in a shared library.

   Currently, this mechanism is *not* publicly supported. However, it is used internally by the following Artix services: LocatorService, SessionManagerService, PeerManager, and ContainerService.

References

For more details about how to register servants, see “Registering Static Servants” on page 52 and “Registering Transient Servants” on page 56.

Registering Static Servants

Initially, when a servant object is created, it is associated with a particular *logical contract* (that is, WSDL port type), but has no association with any *physical contract* (that is, WSDL service). The link between a servant instance and a physical contract must be established explicitly by *registering* the servant.

Figure 8 illustrates the effect of registering a static servant: registration establishes an association between a servant instance and a part of the WSDL model that represents a particular WSDL service.

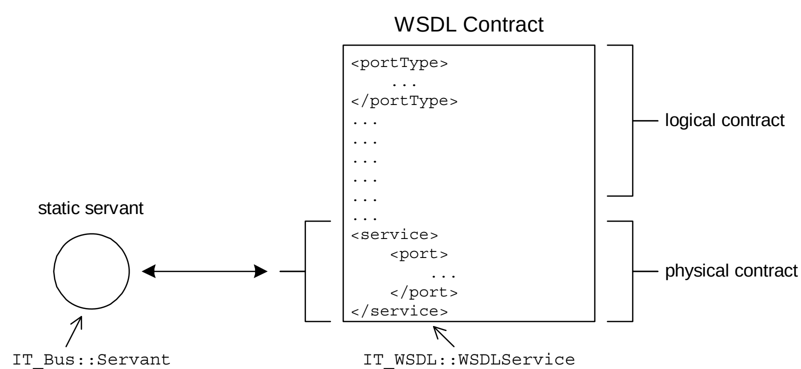

Figure 8: *Relationship between a Static Servant and a WSDL Contract*

Static servant

The defining characteristic of a static servant is that, when registered, it is associated with a service appearing *explicitly* in the original WSDL contract. This implies that a static servant is restricted to using a service from the fixed collection of services appearing in the WSDL contract.

IT_Bus::Bus registration functions

The `IT_Bus::Bus` class defines the functions in Example 20 to manage the registration of static servants:

Example 20: *The IT_Bus::Bus Static Servant Registration API*

```
// C++
void
register_servant(
    IT_Bus::Servant & servant,
    IT_WSDL::WSDLService & wsdl_service,
```

Example 20: *The IT_Bus::Bus Static Servant Registration API*

```
    const IT_Bus::String & port_name = IT_BUS_ALL_PORTS
) IT_THROW_DECL((IT_Bus::Exception)) = 0;

void
register_servant(
    IT_Bus::Servant & servant,
    const IT_Bus::String & wsdl_location,
    const IT_Bus::QName & service_name,
    const IT_Bus::String & port_name = IT_BUS_ALL_PORTS
) IT_THROW_DECL((Exception)) = 0;

IT_Bus::Service_ptr
add_service(
    IT_WSDL::WSDLService & wsdl_service
) IT_THROW_DECL((IT_Bus::Exception)) = 0;

IT_Bus::Service_ptr
add_service(
    const IT_Bus::String & wsdl_location,
    const IT_Bus::QName & service_name
) IT_THROW_DECL((Exception)) = 0;

virtual IT_WSDL::WSDLService*
get_service_contract(
    const QName& service_name
) IT_THROW_DECL((Exception)) = 0;

IT_Bus::Service_ptr
get_service(
    const IT_Bus::QName & service_name
);

void
remove_service(
    const QName & service_name
);
```

IT_Bus::Service registration function

In addition to the registration functions in `IT_Bus::Bus`, the `IT_Bus::Service` class also supports a `register_servant()` function. The `IT_Bus::Service::register_servant()` function enables you to activate ports individually.

Example 21: *The IT_Bus::Service register_servant() Function*

```
// C++
void
register_servant(
    IT_Bus::Servant & servant,
    const IT_Bus::String & port_to_register
);
```

Activating a static servant

There are different approaches to activating a static servant, depending on whether you want to activate ports together or individually. The following approaches are supported:

- Activate all ports together.
- Activate ports individually.

Activate all ports together

To activate all ports together, registration is a single step process. You add the service to the Bus and activate all of its ports by calling `IT_Bus::Bus::register_servant()`. For example:

```
// C++
PlugInImpl::PlugInImpl(
    Bus_ptr bus
) IT_THROW_DECL((Exception))
  :
    BusPlugIn(bus),
    m_bank_servant(bus),
    m_service_qname("", "BankService",
   "http://www.iona.com/bus/demos/bank")
{
    // complete
}

void
PlugInImpl::bus_init(
) IT_THROW_DECL((Exception))
{
    IT_WSDL::WSDLService* wsdl_service =
        get_bus()->get_service_contract(m_service_qname);

    bus->register_servant(
        m_bank_servant,
        *wsdl_service
    );
}

void
PlugInImpl::bus_shutdown(
) IT_THROW_DECL((Exception))
{
    get_bus()->remove_service(m_service_qname);
}
```

In this case, all the service's ports dispatch their invocations to the same servant object, `m_bank_servant`.

Activate ports individually

To activate ports individually, registration is a two-step process. First you add a service to the Bus, then you activate individual ports. For example:

```
// C++
PlugInImpl::PlugInImpl(
    Bus_ptr bus
) IT_THROW_DECL((Exception))
  :
    BusPlugIn(bus),
    m_corba_servant(bus),
    m_soap_servant(bus),
    m_service_qname("", "BankService",
   "http://www.iona.com/bus/demos/bank")
{
    // complete
}

void
PlugInImpl::bus_init(
) IT_THROW_DECL((Exception))
{
    IT_WSDL::WSDLService* wsdl_service =
        get_bus()->get_service_contract(m_service_qname);

    IT_Bus::Service_var bank_service =
        get_bus()->add_service(*wsdl_service);
    bank_service->register_servant(m_corba_servant,"CORBAPort");
    bank_service->register_servant(m_soap_servant, "SOAPPort");
}

void
PlugInImpl::bus_shutdown(
) IT_THROW_DECL((Exception))
{
    get_bus()->remove_service(m_service_qname);
}
```

In this case, each port can be programmed to dispatch invocations to distinct servant objects. For example, invocations arriving at the CORBAPort port are dispatched to the corba_servant servant instance. Whereas, invocations arriving at the SOAPPort port are dispatched to the soap_servant servant instance.

Default threading model

The default threading model for a registered servant is *multi-threaded*. That is, the servant is liable to have its operations invoked simultaneously by multiple threads. With this model, it is essential to ensure that your servant code is reentrant and thread-safe. Alternatively, you can select another threading model when registering the servant.

See “Servant Threading Models” on page 103 for more information.

Registering Default Servants

For information on registering default servants, in "Functions Defined on IT_Bus::Service" on page 39, see the explanation of the `register_default_servant()` member function.

Registering Transient Servants

In contrast to a static servant, a transient servant is not limited to using services that appear explicitly in the WSDL contract. A transient servant creates a new service every time it is registered by *cloning* from an existing service (that is, a *template service*) in the WSDL contract. This behavior is useful in cases where you require an unlimited number of services of a particular kind.

For example, consider the WSDL contract for the `demos/servant_management/transient_servants` demonstration, which has a `Bank` port type and an `Account` port type. In this case, you require an unlimited number of `Account` services to represent customer accounts.

Figure 9 illustrates the effect of registering a transient servant. Registration establishes an association between a servant instance and a cloned service.

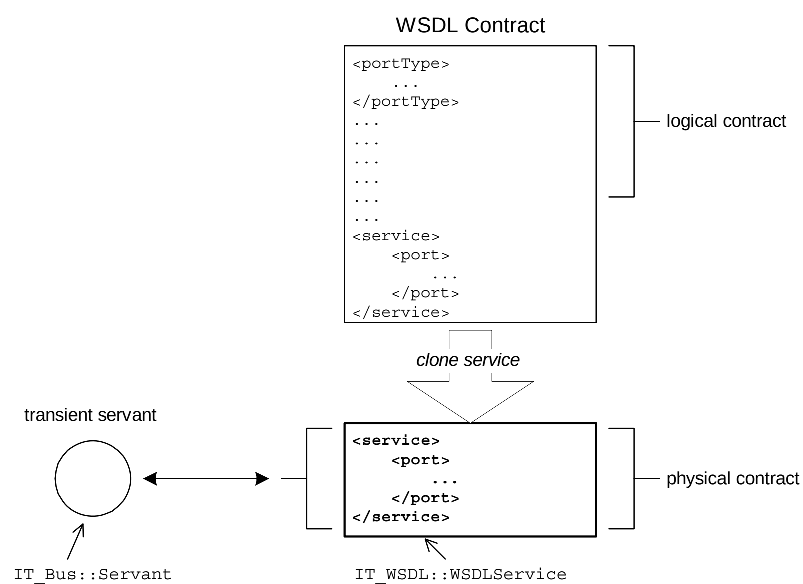

Figure 9: *Relationship between a Transient Servant and a WSDL Contract*

Supported protocols

Artix currently supports transient servants for the following transports:

- HTTP
- CORBA
- Tunnel

Template service

A prerequisite for creating transient services is that you define a *template service* in the WSDL contract. A template service is distinguished by having a port address that is a placeholder (otherwise, the template is like an ordinary `service` element).

For example, the placeholder for a HTTP port address is any URL of the form `http://`*Hostname*`:`*Port* (or `https://`*Hostname*`:`*Port* for a secure service).

Transient servant registration

When a transient servant is registered, the following steps are implicitly performed by the `IT_Bus::Bus` instance (see Figure 9):

1. A new WSDL service is cloned from an existing service in the WSDL contract. The *cloned service* has the following characteristics:
   - The cloned service is based on an existing `service` element that appears in the WSDL contract.
   - The clone's service QName is replaced by a dynamically generated, unique service QName.
   - The clone's addressing information is replaced such that each address is unique per-clone and per-port.
2. The transient servant becomes associated with the newly cloned service.

Examples of transient services

Transient services are currently supported by the HTTP, CORBA and Tunnel transports. For example, you could define the following kinds of template:

- SOAP template service.
- CORBA template service.

SOAP template service

Example 22 shows an example of a SOAP service that could be used as a template for cloning transient SOAP services.

Example 22: *Example of a HTTP Template Service*

```
<service name="ServiceName">
    <port name="PortName" binding="BindingName">
        <soap:address location="http://localhost:0" />
        ...
    </port>
</service>
```

The SOAP template service has the following features:

- The *ServiceName* and *PortName* are the same as the values passed to the `IT_Bus::Bus::register_transient_servant()` function in the application code.
- The `location` attribute of `<soap:address>` must be initialized with a placeholder URL, `http://`*Hostname*`:`*Port*. If the URL has the special form, `http://localhost:0`, Artix substitutes the actual host name and a dynamically allocated IP port.

CORBA template service

Example 23 shows an example of a CORBA service that could be used as a template for cloning transient CORBA services.

Example 23: *Example of a CORBA Template Service*

```
<service name="ServiceName">
    <port name="PortName" binding="BindingName">
        <corba:address location="ior:" />
        ...
    </port>
</service>
```

The CORBA template service has the following features:

- The *ServiceName* and *PortName* are the same as the values passed to the `IT_Bus::Bus::register_transient_servant()` function in the application code.
- The `location` attribute of `<corba:address>` must be initialized with the `ior:` placeholder IOR.

Reuse of IP ports

To avoid over-use of IP ports, cloned services are designed to use the same IP ports as the template service.

IT_Bus::Bus transient registration functions

The `IT_Bus::Bus` class defines the functions in Example 24 to manage the registration of transient servants.

Example 24: *The IT_Bus::Bus Transient Servant Registration API*

```
// C++
IT_Bus::Service_ptr
register_transient_servant(
    IT_Bus::Servant & servant,
    IT_WSDL::WSDLService & wsdl_service,
    const IT_Bus::String & port_name = IT_BUS_ALL_PORTS
) IT_THROW_DECL((IT_Bus::Exception)) = 0;

IT_Bus::Service_ptr
register_transient_servant(
```

Example 24: *The IT_Bus::Bus Transient Servant Registration API*

```
    IT_Bus::Servant & servant,
    const IT_Bus::String & wsdl_location,
    const IT_Bus::QName & service_name,
    const IT_Bus::String & port_name = IT_BUS_ALL_PORTS
) IT_THROW_DECL((Exception)) = 0;

IT_Bus::Service_ptr
add_transient_service(
    IT_WSDL::WSDLService & wsdl_service
) IT_THROW_DECL((IT_Bus::Exception)) = 0;

IT_Bus::Service_ptr
add_transient_service(
    const IT_Bus::String & wsdl_location,
    const IT_Bus::QName & service_name
) IT_THROW_DECL((Exception)) = 0;

virtual IT_WSDL::WSDLService*
get_service_contract(
    const QName& service_name
) IT_THROW_DECL((Exception)) = 0;

IT_Bus::Service_ptr
get_service(
    const IT_Bus::QName & service_name
);

void
remove_service(
    const IT_Bus::QName & service_name
);
```

IT_Bus::Service registration function

In addition to the registration functions in `IT_Bus::Bus`, the `IT_Bus::Service` class also supports a `register_servant()` function. The `IT_Bus::Service::register_servant()` function enables you to activate ports individually.

Example 25: *The IT_Bus::Service register_servant() Function*

```
// C++
void
register_servant(
    IT_Bus::Servant & servant,
    const IT_Bus::String & port_to_register
);
```

Activating a transient servant

There are several different approaches to activating a transient servant, depending on whether you want to activate ports together or individually and depending on whether you want to specify the WSDL contract directly or use the `get_service_contract()` function. The following approaches are supported:

- Activate all ports together.
- Activate ports individually.

Activate all ports together

Registration is a single step process. You add the transient service to the Bus and activate all of its ports by calling `IT_Bus::Bus::register_transient_servant()`. For example:

Example 26: *Activating All Ports Together for a Transient Servant*

```
// C++
void
BankImpl::create_account(
    const IT_Bus::String &account_name,
    WS_Addressing::EndpointReferenceType &_return
) IT_THROW_DECL((IT_Bus::Exception))
{
    // Find the account data for the account_name account and
    // create a servant, account_servant, to represent it.
    ...  // (not shown)

    // Register account_servant as a transient servant and
    // return a reference to it.

    IT_Bus::QName template_service_name("", "AccountService",
   "http://www.iona.com/bus/demos/bank");

    IT_WSDL::WSDLService* wsdl_template_service =
        get_bus()->get_service_contract(template_service_name);

    IT_Bus::Service_var cloned_service =
        get_bus()->register_transient_servant(
            account_servant,
            *wsdl_template_service
        );

    get_bus()->populate_endpoint_reference(
        cloned_service->get_wsdl_service(),
        _return
    );
}
```

In this case, all the service's ports dispatch their invocations to the same servant object, `account_servant`.

Note that the `IT_WSDL::WSDLService` object passed to `register_transient_service()`, `wsdl_template_service`, represents the *template service*, whereas the `IT_Bus::Service` object returned by `register_transient_service()` represents the *cloned service*. When generating the endpoint reference for the transient service (by calling `populate_endpoint_reference()`), you must generate the reference from the cloned service, *not* from the template service.

Activate ports individually

Registration is a two-step process. First you add a transient service to the Bus (thereby cloning the service), and then you activate individual ports. For example:

Example 27: *Activating Ports Individually for a Transient Servant*

```
// C++
void
BankImpl::create_account(
    const IT_Bus::String &account_name,
    WS_Addressing::EndpointReferenceType &_return
) IT_THROW_DECL((IT_Bus::Exception))
{
    // Find the account data for the account_name account and
    // create two servants: corba_servant and soap_servant.
    // These servants provide distinct implementations of the
    // Account service, for the CORBA and SOAP protocols
    // respectively.
    ...  // (not shown)

    // Register account_servant as a transient servant and
    // return a reference to it.

    IT_Bus::QName template_service_name("", "AccountService",
   "http://www.iona.com/bus/demos/bank");

    IT_WSDL::WSDLService* wsdl_template_service =
        get_bus()->get_service_contract(template_service_name);

    IT_Bus::Service_var cloned_service =
        get_bus()->add_transient_service(*wsdl_template_service);
    cloned_service->register_servant(corba_servant,"CORBAPort");
    cloned_service->register_servant(soap_servant, "SOAPPort");

    get_bus()->populate_endpoint_reference(
        cloned_service->get_wsdl_service(),
        _return
    );
}
```

In this case, each port can be programmed to dispatch invocations to distinct servant objects. For example, invocations arriving at the `CORBAPort` port are dispatched to the `corba_servant` servant instance; whereas invocations arriving at the `SOAPPort` port are dispatched to the `soap_servant` servant instance.

Default threading model

The default threading model for a registered servant is *multi-threaded*. That is, the servant is liable to have its operations invoked simultaneously by multiple threads. With this model, it is essential to ensure that your servant code is reentrant and thread-safe. Alternatively, you can select another threading model when registering the servant.

See “Servant Threading Models” on page 103 for more information.

Client Programming

This chapter describes how to develop an Artix client. The key concepts that a client programmer needs to understand are references, which encapsulate the location of a remote service, and client proxies, which enable you to invoke WSDL operations.

Programming with Client Proxies

Client proxies are the basic objects needed for Web services programming on the client side. A client proxy is a C++ object that provides a Remote Procedure Call (RPC) interface to a local or remote Web service. Each proxy instance represents a connection to a particular service endpoint and the proxy's member functions provide programmatic access to the service's WSDL operations.

What is a Client Proxy?

A client proxy is a C++ object that exposes member functions that correspond to WSDL operations from a specific WSDL port type. By calling the C++ functions exposed by the proxy, a client can invoke the corresponding operations on a Web service, either locally or remotely.

Figure 10 illustrates the role of a client proxy in a distributed Web services application. In this example, the client proxy represents a `Greeter` port type, which supports the `sayHi` WSDL operation. When the client calls the `sayHi()` function on the proxy, the proxy converts this call into a request message, which is transmitted to the server port. The server then converts the request message to a `sayHi()` function call on a servant object. The return values from the `sayHi()` call are transmitted back to the client in a reply message.

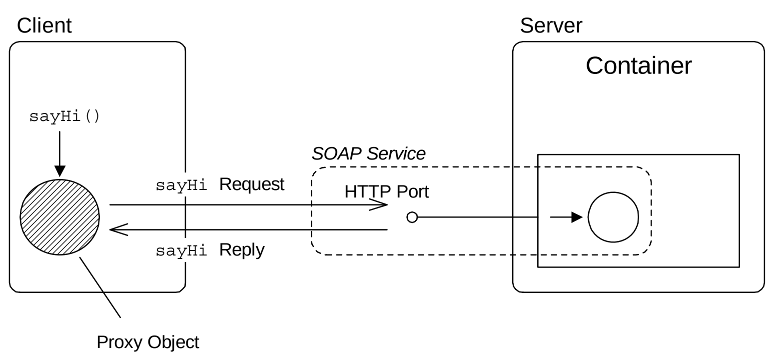

Figure 10: *Role of a Client Proxy in a Distributed Application*

Client proxy features

Artix client proxies provide the following advantages to the client programmer:

- *Location invariance*—calls can be made either on local or remote services. The syntax and semantics are the same in either case.
- *Protocol invariance*—the syntax of client calls is independent of the underlying binding and transport protocol.
- *Distributed exception handling*—exceptions raised in a remote server are automatically propagated back to the client and raised as local exceptions.

Greeter WSDL port type

The interface for a client proxy is defined by a *WSDL port type*. The port type defines a collection of operations which are mapped to C++ functions by the WSDL-to-C++ compiler. For example, Example 28 shows the `Greeter` port type, which defines two WSDL operations, `sayHi` and `greetMe`.

Example 28: *Greeter WSDL Port Type*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloWorld"
    targetNamespace="http://www.iona.com/hello_world_soap_http"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.iona.com/hello_world_soap_http"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <types>
        <schema targetNamespace="http://www.iona.com/hello_world_soap_http"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
            <element name="responseType" type="xsd:string"/>
            <element name="requestType" type="xsd:string"/>
        </schema>
    </types>
    <message name="sayHiRequest"/>
    <message name="sayHiResponse">
        <part element="tns:responseType" name="theResponse"/>
    </message>
    <message name="greetMeRequest">
        <part element="tns:requestType" name="me"/>
    </message>
    <message name="greetMeResponse">
        <part element="tns:responseType" name="theResponse"/>
    </message>

    <portType name="Greeter">
        <operation name="sayHi">
            <input message="tns:sayHiRequest" name="sayHiRequest"/>
            <output message="tns:sayHiResponse" name="sayHiResponse"/>
```

Example 28: *Greeter WSDL Port Type*

```
        </operation>
        <operation name="greetMe">
            <input message="tns:greetMeRequest" name="greetMeRequest"/>
            <output message="tns:greetMeResponse" name="greetMeResponse"/>
        </operation>
    </portType>
    ...
</definitions>
```

Greeter proxy class

To generate a proxy class, run the WSDL-to-C++ compiler with the appropriate options (see “Generating client stub code” on page 72 for details). The proxy class implementation is contained in the *client stub files*. For example, compiling the `Greeter` port type generates the following stub files:

```
Greeter.h
GreeterClient.h
GreeterClient.cxx
```

The generated proxy class, `GreeterClient`, is shown in Example 29.

Example 29: *Generated GreeterClient Proxy Class*

```
  // C++
1 namespace COM_IONA_HELLO_WORLD_SOAP_HTTP
  {
2     class GreeterClient : public Greeter, public
    IT_Bus::ClientProxyBase
      {

        public:
3         // Constructors and Destructor
          // (not shown)
          ...

          virtual void
4         sayHi(
              IT_Bus::String &theResponse
          ) IT_THROW_DECL((IT_Bus::Exception));

          virtual void
          greetMe(
              const IT_Bus::String &me,
              IT_Bus::String &theResponse
          ) IT_THROW_DECL((IT_Bus::Exception));

      };
  }
```

The preceding code example can be explained as follows:

1. By default, the C++ namespace enclosing the proxy class is derived from the target namespace of the corresponding WSDL port type. For example, the `Greeter` port type is defined with the target namespace,

http://www.iona.com/hello_world_soap_http, which translates to the C++ namespace, COM_IONA_HELLO_WORLD_SOAP_HTTP. It is also possible to override the default namespace name.

2. In general, a proxy class generated from the *PortTypeName* port type maps to a C++ class, *PortTypeName*Client. For example, the Greeter port type maps to the C++ class, GreeterClient.
3. In general, you must specify the protocol and connection details when initializing a client proxy instance. The proxy class itself is completely protocol-independent.

   The proxy constructors are not shown here—for a discussion of proxy constructors, see "Initializing Proxies from References" on page 67 and "Other Ways of Initializing Proxies" on page 70.
4. The proxy class includes C++ member functions that correspond to each of the WSDL operations defined in the Greeter port type.

WSDL services for the proxy

Apart from representing a WSDL port type, each instance of a client proxy encapsulates specific protocol and connection details, which correspond to the information in a WSDL service element. Thus, a WSDL service element effectively represents the state of a proxy object.

Example 30 shows a WSDL service with a single port. In this case, the HelloWorldSOAPService service unambiguously represents a single endpoint.

Example 30: *WSDL Service with Single Port*

```
<definitions
   targetNamespace="http://www.iona.com/hello_world_soap_http"
   ... >
    ...
    <service name="HelloWorldSOAPService">
        <port binding="tns:GreeterSOAPBinding" name="HTTPPort">
            <http:address location="http://localhost:4444"/>
        </port>
    </service>
</definitions>
```

Example 31 shows a WSDL service with multiple ports. In this case, the MultiPortService service represents two different endpoints. In order to choose which endpoint to connect to, you must use a form of proxy constructor that lets you specify the port name. See “Initializing Proxies from References” on page 67 and “Other Ways of Initializing Proxies” on page 70 for details.

Example 31: *WSDL Service with Multiple Ports*

```
<definitions
   targetNamespace="http://www.iona.com/hello_world_soap_http"
   ... >
    ...
    <service name="MultiPortService">
        <port binding="tns:GreeterSOAPBinding" name="HTTPPort">
            <http:address location="http://localhost:3333"/>
        </port>

        <port binding="tns:GreeterGIOPBinding" name="IIOPPort">
            <corba:address
                location="file:../../hello_world_service.ior"/>
        </port>
    </service>
</definitions>
```

Initializing Proxies from References

Typically, the cleanest way to initialize a client proxy is by constructing it from an endpoint reference. A reference object encapsulates all of the information needed to open a connection to a particular service. By using references in your client program, it is relatively easy to avoid hard-coding details such as the location of a WSDL contract file.

This subsection describes both how to use references to initialize proxies and how to obtain the references themselves.

Proxy constructors with a reference argument

To initialize a proxy from a reference, the `GreeterClient` class defines the constructors shown in Example 32.

Example 32: *Proxy Constructors with a Reference Argument*

```
GreeterClient(
    const WS_Addressing::EndpointReferenceType & epr_ref
);

GreeterClient(
    const WS_Addressing::EndpointReferenceType& epr_ref,
    const IT_Bus::String&                       wsdl_location,
    const IT_Bus::QName&                        service_name,
    const IT_Bus::String&                       port_name
);
```

Constructor with a reference argument

The first constructor takes one argument representing an endpoint reference, `WS_Addressing::EndpointReferenceType`. The endpoint reference contains complete service and port details, including addressing information, enabling the client proxy to open a connection to a remote service. This form of constructor is suitable for a reference that contains details of just a single WSDL port.

For a detailed discussion of endpoint references, see "Endpoint References" on page 119.

Constructor with reference argument and contract details

The second constructor takes additional arguments—`wsdl_location`, `service_name`, and `port_name`—that can provide additional information about the endpoint. This constructor is useful in the following cases:

- *The endpoint reference contains multiple ports*—in this case you can use the `port_name` argument to specify which port the client connects to, while leaving the `wsdl_location` and `service_name` arguments empty.

  For example, to initialize a proxy that connects to the `CORBAPort` port from the `multi_port_epr` endpoint reference:

```
// C++
AccountClient* proxy = new AccountClient(
    multi_port_epr,
    IT_Bus::String::EMPTY,
    IT_Bus::QName::EMPTY_QNAME,
    "CORBAPort"
);
```

- *The endpoint reference lacks metadata*—when a reference originates from a non-Artix service, sometimes it might contain just an URL (the endpoint address) and provide no other details about the endpoint. In this case, you can supply the missing endpoint details from a WSDL contract, by specifying the WSDL contract location, `wsdl_location`, the service QName, `service_name`, and port name, `port_name`, for the endpoint.

Obtaining a reference

You can obtain an endpoint reference from one of the following sources:

- Initial reference mechanism.
- Return value from a WSDL operation.
- Artix locator.

Initial reference mechanism

The Artix *initial reference mechanism* provides a layer of abstraction for obtaining references. The client programmer requests a reference to a particular WSDL service, by passing the service's QName to the `IT_Bus::Bus::resolve_initial_references()` function. The source of the WSDL service description is determined independently of this function call. For example, the location of a file containing a WSDL service might be provided as a command-line argument to the client executable.

The function for obtaining an initial reference has the following signature:

```
// C++
// In IT_Bus::Bus
virtual IT_Bus::Boolean resolve_initial_reference(
    const IT_Bus::QName & service_name,
    WS_Addressing::EndpointReferenceType &
   endpoint_reference
) IT_THROW_DECL((Exception)) = 0;
```

For more details, see "Programming with Initial References" on page 74.

Return value from a WSDL operation

Endpoint references can be passed as parameters in WSDL operations. Hence, a common way of obtaining a reference is as a return value from a WSDL operation.

For example, consider a Bank service that manages customer accounts. The Bank service could provide a WSDL operation, `get_account`, that returns a reference to an Account service. You could define the `get_account` operation as follows:

```
<definitions ... >
    ...
    <message name="get_account">
        <part name="account_name" type="xsd:string"/>
    </message>
    <message name="get_accountResponse">
        <part name="return" type="wsa:EndpointReferenceType"/>
    </message>
    ...
    <portType name="Bank">
        ...
        <operation name="get_account">
            <input name="get_account" message="tns:get_account"/>
            <output name="get_accountResponse" message="tns:get_accountResponse"/>
        </operation>
        ...
    </portType>
</definitions>
```

In the Bank proxy class, the `get_account` operation would map to a C++ function, `get_account()`, as follows:

```
// C++
void get_account(
    const IT_Bus::String &account_name,
    WS_Addressing::EndpointReferenceType &_return
) IT_THROW_DECL((IT_Bus::Exception));
```

The return value from `get_account()` is represented by the `WS_Addressing::EndpointReferenceType` type. For more details, see "Endpoint References" on page 119.

Artix locator

The Artix locator is a dedicated service for storing and retrieving references. The mechanism for retrieving references from the locator consists essentially of calling a WSDL operation that returns a reference. For more details about the Artix locator service, see the *Artix Locator Guide*.

Other Ways of Initializing Proxies

Instead of initializing a proxy using an endpoint reference, you can specify the proxy's connection information explicitly: WSDL location URL, service QName, and port name. This way of initializing a proxy is useful, if you need to provide the proxy's connection information in a customized manner.

Other proxy constructors

Besides the constructors with reference arguments (see Example 32 on page 67), the `GreeterClient` class defines the constructors shown in Example 33.

Example 33: *Other Proxy Constructors*

```
GreeterClient();

GreeterClient(
    const IT_Bus::String & wsdl
);

GreeterClient(
    const IT_Bus::String & wsdl,
    const IT_Bus::QName & service_name,
    const IT_Bus::String & port_name
);
```

Constructor with no arguments

When using the constructor with no arguments, the client requires that the contract defining its behavior be located in the same directory as the executable. The client uses the service name specified at code generation time using the `-e` flag.

If the specified service has multiple WSDL ports, the client proxy connects by default to the first port in the `wsdl:service` element.

Constructor with WSDL URL argument

The second constructor takes one argument that allows you to specify the URL of the contract defining the client's behavior. The client uses the service specified at code generation time using the `-e` flag.

In particular, the `wsdl` argument could be a `file:` URL or a `uddi:` URL (for details of how to use UDDI, see "Locating Services with UDDI" on page 114).

Constructor with WSDL URL, service, and port arguments

The fourth constructor provides you with the most flexibility in determining how the client connects to its server. It takes three arguments:

`wsdl`	Specifies the URL of the contract defining the client's behavior.
`service_name`	Specifies the QName of the service, defined in the contract with a `<service>` tag, to use when connecting to the server.
`port_name`	Specifies the name of the port, defined in the contract with a `<port>` tag, to use when connecting to the server. The port name given must be defined in the specified `<service>` tag. If you don't want to specify the port name, you can leave this argument blank by passing `IT_Bus::String::EMPTY`. In this case, the client proxy connects to the first port in the `wsdl:service` element.

The ability to specify the port name in the constructor is useful for WSDL services that contain multiple ports—for example, see Example 31 on page 67. This argument enables you to pick one of the ports explicitly, instead of defaulting to the first port in the `service` element.

Implementing a Client

The stub code for a client implementation of the service defined by the contract is contained in the files *PortTypeName*`Client.h` and *PortTypeName*`Client.cxx`. You should never make any modifications to the generated code in these files.

To access the operations defined in the port type, the client initializes the Artix bus, instantiates an object of the generated client proxy class, `*PortTypeName*Client`, and makes function calls on the object. When the client is finished, it then shuts down the bus.

Generating client stub code

To generate client stub code from the Hello World WSDL contract, `hello_world.wsdl`, enter the following command (for your respective platform):

Windows

```
wsdltocpp -i Greeter
   -e HelloWorldSOAPService
   -client
   -sample
   -m NMAKE:executable
   hello_world.wsdl
```

UNIX

```
wsdltocpp -i Greeter
   -e HelloWorldSOAPService
   -client
   -sample
   -m UNIX:executable
   hello_world.wsdl
```

The `-client` switch ensures that client stub code is generated. For full details of the `wsdltocpp` switches, see "Generating code from the command line" on page 484.

Initializing the Bus

Client applications initialize the Bus, by calling `IT_Bus::init()`. You should always pass the command-line arguments from `main()` to `IT_Bus::init()`. This ensures that you can use standard Artix switches at the command-line (for example, `-BUSname` *BusID* to specify the Bus ID at the command line).

Invoking the operations

To invoke the operations offered by the service, the client calls the member functions of the client proxy object. The generated client proxy class contains one member function for each operation defined in the contract. The generated functions all return void. Any response messages are passed by reference as a parameter to the function. For example, the `greetMe` operation defined in Example 28 on page 64 generates a function with the following signature:

```
void greetMe(
    const IT_Bus::String & me,
    IT_Bus::String & var_return
) IT_THROW_DECL((IT_Bus::Exception));
```

Full client code

A client developed to access the service defined by the `HelloWorldSOAPService` contract will look similar to Example 34.

Example 34: *Sample Hello World Client*

```
// C++
#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_cal/iostream.h>

1 #include "GreeterClient.h"

2 IT_USING_NAMESPACE_STD

3 using namespace COM_IONA_HELLO_WORLD_SOAP_HTTP;
using namespace IT_Bus;

int
main(
    int argc,
    char* argv[]
)
{
    cout << " GreeterClient" << endl;

    try
    {
        /*
         *  Create an instance of the web service client
         */
4       IT_Bus::init(argc, argv);

5       GreeterClient client;

        // Sample invocation calls.
        //
        IT_Bus::String theResponse;
6       client.sayHi(theResponse);
        cout << "sayHi() returned: \"" << theResponse << "\""
             << endl;

        IT_Bus::String me = "YourName";
        client.greetMe(me, theResponse);
        cout << "greetMe() returned: \"" << theResponse << "\""
             << endl;
    }
7   catch(IT_Bus::Exception& e)
```

Example 34: *Sample Hello World Client*

```
    {
        cout << endl << "Error : Unexpected error occured!"
             << endl << e.message()
             << endl;
        return -1;
    }
    return 0;
}
```

The preceding code can be explained as follows:

1. The *PortName*`Client.h` header includes the definitions for the client proxy class.
2. The `IT_USING_NAMESPACE_STD` preprocessor macro expands to the following line of code:

   ```
   // C++
   using namespace std;
   ```

   The `std` namespace scopes entities from the C++ Standard Template Library. For example, using this namespace lets you write `cout` and `cin`, instead of `std::cout` and `std::cin`.
3. The `COM_IONA_HELLO_WORLD_SOAP_HTTP` namespace contains the client proxy class, `GreeterClient`. See “Greeter proxy class” on page 65.
4. The `IT_Bus::init()` static function initializes the bus. You should always pass in the command line arguments (`argc` and `argv`) to `init()`.
5. This line instantiates the proxy class using the no-argument form of the proxy client constructor. When this client is deployed, a copy of the contract defining its behavior must be deployed in the same directory as the client executable.

   In a real application, however, it would be better to initialize the client proxy from an initial reference. See “Programming with Initial References” on page 74.
6. Invoke the `sayHi()` operation on the client proxy.
7. Catch any exceptions thrown by the bus. It is essential to enclose remote operation invocations within a try/catch block which catches the exception types derived from `IT_Bus::Exception`.

Programming with Initial References

Artix provides an API function, `IT_Bus::resolve_initial_references()`, for finding endpoint references based on the service QName.

The initial reference mechanism abstracts the procedure for obtaining endpoint references. Using this approach, a programmer needs to know only the *name of a service* in order to create a proxy. The endpoint details could actually be provided from configuration, from the command-line, by programming, or by some other method. The client programmer does not have to worry about the precise source of the endpoint reference.

Order of precedence for initial reference sources

Artix finds initial references from the following sources, in order of priority:

1. *Colocated service*—if the client code that calls `resolve_initial_reference()` is colocated with (that is, in the same process as) the required service, the `resolve_initial_reference()` function returns a reference to the colocated service. This assumes that the client and server code are using the same Bus instance.
2. *References registered using* `register_initial_reference()`—you can register a reference explicitly by calling the `IT_Bus::Bus::register_initial_reference()` function on a Bus instance.
3. *References specified on the command line*—you can provide an initial reference by specifying on the command line the location of a file containing an endpoint reference. For example:

   ```
   GreeterClient -BUSname BusID
      -BUSinitial_reference ../../etc/hello_ref.xml
   ```
4. *References specified in the configuration file*—you can provide an initial reference from the configuration file, either by specifying the location of an endpoint reference file or by specifying the literal value of an endpoint reference.

   For more details, see *Configuring and Deploying Artix Solutions*.
5. *Service in a WSDL contract*—the `service` element in a WSDL contract contains essentially the same data as an endpoint reference. Hence, if a reference is not specified using one of the other methods, Artix searches any loaded WSDL contracts to find the specified service.

   The sources of WSDL contracts are the same as on the server side. The mechanism for finding references is, thus, effectively an extension of the mechanism for finding WSDL contracts—see "How Services Locate WSDL Contracts" on page 50.

Example of programming with an initial reference

Given that the Bus has already loaded and parsed the details of a service called `HelloWorldSOAPService` in the namespace, `http://www.iona.com/hello_world_soap_http`, you can initialize a client proxy, `proxy`, as follows:

Example 35: *Resolving an Initial Reference*

```
// C++
IT_Bus::QName service_qname(
  "", "HelloWorldSOAPService", "http://www.iona.com/hello_world_soap_http"
);
WS_Addressing::EndpointReferenceType ref;
```

Example 35: *Resolving an Initial Reference*

```
// Find the initial reference using the bootstrap service
bus->resolve_initial_reference(
    service_qname,
    ref
);

// Create a proxy and use it
GreeterClient proxy(ref);
proxy.sayHi();
```

Abbreviated constructor for initial references

To simplify the steps required to create a proxy from an initial reference, Artix provides a special constructor that initializes a proxy from a service QName in a single step. The constructor has the following form (for a `GreeterClient` proxy):

```
GreeterClient(
    const IT_Bus::QName     service_name,
    const IT_Bus::String&  port_name = IT_Bus::String::EMPTY,
    IT_Bus::Bus_ptr        bus = 0
);
```

With this constructor, you can initialize a proxy from an initial reference using the code fragment shown in Example 36.

Example 36: *Resolving an Initial Reference with a Special Constructor*

```
// C++
IT_Bus::QName service_qname(
  "", "HelloWorldSOAPService", "http://www.iona.com/hello_world_soap_http"
);

// Create a proxy and use it
GreeterClient proxy(service_qname);
proxy.sayHi();
```

Where the proxy constructor implicitly looks up the initial reference based on the specified service QName, `service_qname`.

Obtaining Initial References

Given that you have programmed your client to use initial references, as described in the previous section, you then need provide those initial references at runtime. This section describes how to obtain the initial references needed by the client and how to pass the initial references to the client through its command-line arguments.

Options for obtaining initial references

Some of the possible options for obtaining initial references are, as follows:

- Access local WSDL contract.
- Obtain reference from a container.
- Obtain WSDL contract from a container.
- Obtain WSDL location URL from a container.

Access local WSDL contract

If a WSDL service uses a *statically allocated port* (where the IP port is specified explicitly in the original WSDL contract), the client can obtain the endpoint reference from a local copy of the WSDL contract. When using the initial references API, you can specify the location of the WSDL contract using the command-line switch, `-BUSservice_contract` *WSDLFile*, where *WSDLFile* is a WSDL contract that provides initial references for the client. For example, you can run the Greeter client as follows:

```
GreeterClient -BUSname BusID -BUSservice_contract WSDLFile
```

Obtain reference from a container

You can obtain an endpoint reference directly from an Artix container, after the container has started up. Use the `it_container_admin` utility to retrieve the endpoint reference and store it in a file, as follows:

```
it_container_admin -container ContainerURLFile
   -publishreference
   -service {Namespace}LocalPart
   -file ReferenceFile
```

Where *ContainerURLFile* is a file that contains the URL for the container service (to get this URL file, start `it_container` with the `-publish` option). The service QName is specified by an open brace, {, followed by the target namespace, *Namespace*, followed by a close brace, }, followed by the local part of the service's name, *LocalPart*. For example, the QName for the `HelloWorldSOAPService` service (see Example 30 on page 66) would be specified as follows:

```
{http://www.iona.com/hello_world_soap_http}HelloWorldSOAPService
```

Given that the reference has been stored in the file, *ReferenceFile*, and assuming that the client has access to the file system where this file is stored, you can run the Greeter client as follows:

```
GreeterClient -BUSname BusID -BUSinitial_reference ReferenceFile
```

Obtain WSDL contract from a container

You can obtain a WSDL contract directly from an Artix container, after the container has started up. Use the `it_container_admin` utility to retrieve the WSDL contract and store it in a file, as follows:

```
it_container_admin -container ContainerURLFile
   -publishwsdl
   -service {Namespace}LocalPart
   -file WSDLFile
```

Given that the WSDL contract has been stored in the file, *WSDLFile*, and assuming that the client has access to the file system where this file is stored, you can run the Greeter client as follows:

```
GreeterClient -BUSname BusID -BUSservice_contract WSDLFile
```

Obtain WSDL location URL from a container

You can provide the client with a URL from which the client can download an up-to-date copy of the WSDL contract. Use the `it_container_admin` utility to retrieve the WSDL location URL and store it in a file, as follows:

```
it_container_admin -container ContainerURLFile
   -publishurl
   -service {Namespace}LocalPart
   -file WSDL_URLFile
```

Given that the URL has been stored in the file, *WSDL_URLFile*, and assuming that the client has access to the file system where this file is stored, you can run the Greeter client as follows:

```
GreeterClient -BUSname BusID -BUSservice_contract WSDL_URLFile
```

Overriding a HTTP Address in a Client

Usually, client applications obtain the HTTP address for a remote Web service by parsing the `port` element of a WSDL contract. Sometimes, however, you might need to specify the HTTP address by programming, thereby overriding the value from the WSDL `port` element.

This section describes how to program an Artix client to override the HTTP address, by setting the `HTTP_ENDPOINT_URL` context value.

HTTP address in a WSDL contract

Example 37 shows how to specify the HTTP address in a WSDL contract for a SOAP/HTTP service. The `location` attribute in the `soap:address` element specifies that the `SOAPService` service is running on the `localhost` host and listening on IP port `9000`. By default, clients will use this address, `http://localhost:9000`, to contact the remote `SOAPService`. It is possible, however, to override this address by programming.

Example 37: *HTTP Address Specified in a WSDL Contract*

```
<wsdl:definitions name="HelloWorld"
   targetNamespace="http://www.iona.com/hello_world_soap_http"
    ...>
    <wsdl:service name="SOAPService">
        <wsdl:port binding="tns:Greeter_SOAPBinding"
                   name="SoapPort">
            <soap:address location="http://localhost:9000"/>
            <http-conf:client/>
            <http-conf:server/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

HTTP_ENDPOINT_URL context

You can use the `HTTP_ENDPOINT_URL` context to program the HTTP address that a client uses to contact a Web service, thereby overriding the value configured in the WSDL contract. The mechanism for setting the `HTTP_ENDPOINT_URL` value is based on Artix contexts (see "Artix Contexts" on page 153). The programming steps for overriding the HTTP address are as follows:

1. Obtain a reference to a request context container (of `IT_Bus::ContextContainer` type).
2. Use the request context container to set the `HTTP_ENDPOINT_URL` context.
3. Create a client proxy and invoke an operation on the proxy.

   For the first invocation, Artix takes the address in the `HTTP_ENDPOINT_URL` context and uses it to establish a connection to the remote service. Subsequent invocations on the proxy continue to send requests to the same endpoint address.
4. After the first invocation on the proxy, Artix clears the `HTTP_ENDPOINT_URL` context. Hence, subsequent client proxies created in this thread revert to using the HTTP address configured in the WSDL contract.

How to override the HTTP address

Example 38 shows how to override the HTTP address to contact a `SOAPService` service running on the host, `yourhost`, and IP port, `5432`.

Example 38: *Using HTTP_ENDPOINT_URL to Override a HTTP Address*

```
// C++
#include <it_bus_pdk/context.h>
#include <it_bus_pdk/context_attrs/context_constants.h>
```

Example 38: *Using HTTP_ENDPOINT_URL to Override a HTTP Address*

```
using namespace IT_Bus;
using namespace IT_ContextAttributes;

ContextRegistry* context_registry =
    bus->get_context_registry();

ContextCurrent& context_current =
    context_registry->get_current();

ContextContainer* request_contexts =
    context_current.request_contexts();

IT_Bus::AnyType* any_string = request_contexts->get_context(
    IT_ContextAttributes::HTTP_ENDPOINT_URL,
    true
);

IT_Bus::StringHolder* str_holder =
   dynamic_cast<IT_Bus::StringHolder*>(any_string);

str_holder->set("http://yourhost:5432");

// Open a connection to the SOAPService service at
   yourhost:5432.
GreeterClient hw;
hw.sayHi("Hello World!");
```

The steps for obtaining a reference to a request context follow a standard pattern. For full details about how to program with contexts, see "Artix Contexts" on page 153.

Artix Programming Considerations

Several areas must be considered when programming complex Artix applications.

Operations and Parameters

This section describes how to declare a WSDL operation and how the operation and its parameters are mapped to C++ by the Artix WSDL-to-C++ compiler.

RPC/Literal Style

This subsection describes the RPC/literal style for defining WSDL operations and parameters. The RPC binding style is distinguished by the fact that it uses multi-part messages (one part for each parameter).

For example, the request message for an operation with three input parameters might be defined as follows:

```
<message name="operationRequest">
    <part name="X" type="X_Type"/>
    <part name="Y" type="Y_Type"/>
    <part name="Z" type="Z_Type"/>
</message>
```

Parameter direction in WSDL

WSDL operation parameters can be sent either as *input parameters* (that is, in the client-to-server direction or as *output parameters* (that is, in the server-to-client direction). Hence, the following kinds of parameter can be defined:

- *in parameter*—declared as an input parameter, but not as an output parameter.
- *out parameter*—declared as an output parameter, but not as an input parameter.
- *inout parameter*—declared both as an input and as an output parameter.

How to declare WSDL operations in RPC/literal style

You can declare a WSDL operation in RPC/literal style as follows:

1. Declare a multi-part input message, including all of the in and inout parameters for the new operation (for example, the `testParams` message in Example 39 on page 82).
2. Declare a multi-part output message, including all of the out and inout parameters for the operation (for example, the `testParamsResponse` message in Example 39 on page 82).
3. Within the scope of `<portType>`, declare a single operation which includes a single input message and a single output message.

WSDL declaration of testParams

Example 39 shows an example of a simple operation, `testParams`, which takes two input parameters, `inInt` and `inoutInt`, and two output parameters, `inoutInt` and `outFloat`.

Example 39: *WSDL Declaration of the testParams Operation*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ...>
    ...
    <message name="testParams">
        <part name="inInt" type="xsd:int"/>
        <part name="inoutInt" type="xsd:int"/>
    </message>
    <message name="testParamsResponse">
        <part name="inoutInt" type="xsd:int"/>
        <part name="outFloat" type="xsd:float"/>
    </message>
    ...
    <portType name="BasePortType">
        <operation name="testParams">
            <input message="tns:testParams" name="testParams"/>
            <output message="tns:testParamsResponse"
                    name="testParamsResponse"/>
        </operation>
    ...
</definitions>
```

C++ mapping of testParams

Example 40 shows how the preceding WSDL `testParams` operation (from Example 39 on page 82) maps to C++.

Example 40: *C++ Mapping of the testParams Operation*

```
// C++
void testParams(
    const IT_Bus::Int inInt,
    IT_Bus::Int & inoutInt,
    IT_Bus::Float & outFloat
) IT_THROW_DECL((IT_Bus::Exception));
```

Mapped parameters

When the `testParams` WSDL operation maps to C++, the resulting `testParams()` C++ function signature starts with the in and inout parameters, followed by the out parameters. The parameters are mapped as follows:

- in parameters—are passed by value and declared `const`.
- inout parameters—are passed by reference.
- out parameters—are passed by reference.

WSDL declaration of testReverseParams

Example 41 shows an example of an operation, `testReverseParams`, whose parameters are listed in the opposite order to that of the preceding `testParams` operation.

Example 41: *WSDL Declaration of the testReverseParams Operation*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ...>
    ...
    <message name="testReverseParams">
        <part name="inoutInt" type="xsd:int"/>
        <part name="inInt" type="xsd:int"/>
    </message>
    <message name="testReverseParamsResponse">
        <part name="outFloat" type="xsd:float"/>
        <part name="inoutInt" type="xsd:int"/>
    </message>
    ...
    <portType name="BasePortType">
        <operation name="testReverseParams">
            <output message="tns:testReverseParamsResponse"
                    name="testReverseParamsResponse"/>
            <input message="tns:testReverseParams"
                   name="testReverseParams"/>
        </operation>
    ...
</definitions>
```

C++ mapping of testReverseParams

Example 42 shows how the preceding WSDL testReverseParams operation (from Example 41 on page 83) maps to C++.

Example 42: *C++ Mapping of the testReverseParams Operation*

```
// C++
void testReverseParams(
    IT_Bus::Int &      inoutInt
    const IT_Bus::Int inInt,
    IT_Bus::Float &    outFloat,
) IT_THROW_DECL((IT_Bus::Exception));
```

Order of in, inout and out parameters

In C++, the order of the in and inout parameters in the function signature is the same as the order of the parts in the input message. The order of the out parameters in the function signature is the same as the order of the parts in the output message.

Note: The parameter order is not affected by the relative order of the <input> and <output> tags in the declaration of <operation>. In the mapped C++ signature, the in and inout parameters always appear before the out parameters.

Document/Literal Wrapped Style

This subsection describes the document/literal wrapped style for defining WSDL operations and parameters. The document/literal wrapped style is distinguished by the fact that it uses single-part messages. The single part is defined as a schema element which contains a sequence of elements, one for each parameter.

Request message format

The request message for an operation with three input parameters might be defined as follows:

```
<types>
    <schema>
        <element name="OperationName">
            <complexType>
                <sequence>
                    <element name="X" type="X_Type"/>
                    <element name="Y" type="Y_Type"/>
                    <element name="Z" type="Z_Type"/>
                </sequence>
            </complexType>
        </element>
    </schema>
</types>
<message name="operationRequest">
    <part name="parameters" element="OperationName"/>
</message>
```

The request message in document/literal wrapped style must obey the following conventions:

- The single element that wraps the input parameters must have the same name as the WSDL operation, *OperationName*.
- The single part must have the name, `parameters`.

Reply message format

The reply message for an operation with three output parameters might be defined as follows:

```
<types>
    <schema>
        <element name="OperationNameResult">
            <complexType>
                <sequence>
                    <element name="Z" type="Z_Type"/>
                    <element name="A" type="A_Type"/>
                    <element name="B" type="B_Type"/>
                </sequence>
            </complexType>
        </element>
    </schema>
</types>
<message name="operationReply">
    <part name="parameters" element="OperationNameResult"/>
</message>
```

The reply message in document/literal wrapped style must obey the following conventions:

- The single element that wraps the output parameters must have the form, *OperationName*`Result`.
- The single part must have the name, `parameters`.

How to declare WSDL operations in document/literal wrapped style

You can declare a WSDL operation in document/literal wrapped style as follows:

1. In the `<schema>` section of the WSDL, define an element (the *input part wrapping element*) as a sequence type containing elements for each of the in and inout parameters (for example, the `testParams` element in Example 43 on page 86).
2. In the `<schema>` section of the WSDL, define another element (the *output part wrapping element*) as a sequence type containing elements for each of the inout and out parameters (for example, the `testParamsResult` element in Example 43 on page 86).
3. Declare a single-part input message, including all of the in and inout parameters for the new operation (for example, the `testParams` message in Example 43 on page 86).
4. Declare a single-part output message, including all of the out and inout parameters for the operation (for example, the `testParamsResult` message in Example 43 on page 86).
5. Within the scope of `<portType>`, declare a single operation which includes a single input message and a single output message.

WSDL declaration of testParams in document/literal wrapped style

Example 39 shows an example of a simple operation, `testParams`, which takes two input parameters, `inInt` and `inoutInt`, and two output parameters, `inoutInt` and `outFloat`.

Example 43: *testParams Operation in Document/Literal Wrapped Style*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >
    <wsdl:types>
        <schema targetNamespace="..."
                xmlns="http://www.w3.org/2001/XMLSchema">
            <element name="testParams">
                <complexType>
                    <sequence>
                      <element name="inInt" type="xsd:int"/>
                      <element name="inoutInt" type="xsd:int"/>
                    </sequence>
                </complexType>
            </element>
            <element name="testParamsResult">
                <complexType>
                  <sequence>
                    <element name="inoutInt" type="xsd:int"/>
                    <element name="outFloat" type="xsd:float"/>
                  </sequence>
                </complexType>
            </element>
        </schema>
```

Example 43: *testParams Operation in Document/Literal Wrapped Style*

```
    </wsdl:types>
    <message name="testParams">
        <part name="parameters" element="tns:testParams"/>
    </message>
    <message name="testParamsResult">
       <part name="parameters" element="tns:testParamsResult"/>
    </message>
    <wsdl:portType name="BasePortType">
        <wsdl:operation name="testParams">
            <wsdl:input  message="tns:testParams"
                         name="testParams"/>
            <wsdl:output message="tns:testParamsResult"
                         name="testParamsResult"/>
        </wsdl:operation>
    </wsdl:portType>
    ...
</definitions>
```

C++ default mapping of testParams

The Artix WSDL-to-C++ compiler automatically detects when you use document/literal wrapped style (as long as the WSDL obeys the conventions described here). If document/literal wrapped style is detected, the WSDL-to-C++ compiler (by default) unwraps the operation parameters to generate a normal function signature in C++.

For example, Example 44 shows how the preceding WSDL `testParams` operation (from Example 43 on page 86) maps to C++.

Example 44: *C++ Mapping of the testParams Operation*

```
// C++
void testParams(
    const IT_Bus::Int inInt,
    IT_Bus::Int & inoutInt,
    IT_Bus::Float & outFloat
) IT_THROW_DECL((IT_Bus::Exception));
```

C++ mapping of testParams using -wrapped flag

If you want to disable the auto-unwrapping feature of the WSDL-to-C++ compiler, you can do so by running `wsdltocpp` with the `-wrapped` flag. For example, assuming that the WSDL from Example 43 on page 86 is stored in the `test_params.wsdl` file, you can generate C++ without auto-unwrapping by entering the following at the command line:

```
wsdltocpp -wrapped test_params.wsdl
```

Example 45 shows the result of mapping the WSDL `testParams` operation to C++ with the `-wrapped` flag:

Example 45: *C++ Mapping Using the -wrapped Flag*

```
// C++
virtual void
testParams(
    const testParams &parameters,
    testParamsResult &parameters_1
) IT_THROW_DECL((IT_Bus::Exception));
```

Exceptions

Artix provides a variety of built-in exceptions, which can alert users to problems with network connectivity, parameter marshaling, and so on. In addition, Artix allows users to define their own exceptions, which can be propagated across the network by declaring fault exceptions in WSDL.

System Exceptions

When an error occurs during an operation invocation, Artix throws an exception of `IT_Bus::FaultException` type (which inherits from the `IT_Bus::Exception` base class). The `IT_Bus::FaultException` member functions enable you to access a considerable amount of information about the exception.

IT_Bus::FaultException attributes

A `FaultException` instance has several attributes that provided detailed information about the exception. The following `FaultException` attributes are available:

- *description*—a human-readable string that summarizes the error.
- *category*—a formal category that indicates what kind of error occurred. The following categories are supported:
    - `IT_Bus::FaultCategory::NO_PERMISSION`
    - `IT_Bus::FaultCategory::CONNECTION_FAILURE`
    - `IT_Bus::FaultCategory::MARSHAL_ERROR`
    - `IT_Bus::FaultCategory::NOT_EXIST`
    - `IT_Bus::FaultCategory::TRANSIENT`
    - `IT_Bus::FaultCategory::UNKNOWN`
    - `IT_Bus::FaultCategory::TIMEOUT`
    - `IT_Bus::FaultCategory::VERSION_ERROR`
    - `IT_Bus::FaultCategory::NOT_UNDERSTOOD`
    - `IT_Bus::FaultCategory::MEMORY`
    - `IT_Bus::FaultCategory::BAD_OPERATION`
    - `IT_Bus::FaultCategory::INTERNAL`
    - `IT_Bus::FaultCategory::INVALID_REFERENCE`
    - `IT_Bus::FaultCategory::NOT_IMPLEMENTED`
    - `IT_Bus::FaultCategory::LICENSE`

- *source*—indicates whether the error occurred on the client side or on the server side. The following values are supported:
  - `IT_Bus::FaultSource::CLIENT`
  - `IT_Bus::FaultSource::SERVER`
  - `IT_Bus::FaultSource::UNKNOWN`
- *completion status*—indicates whether or not the operation completed its work on the server side. The following values are supported:
  - `IT_Bus::CompletionStatus::YES`
  - `IT_Bus::CompletionStatus::NO`
  - `IT_Bus::CompletionStatus::MAYBE`

IT_Bus::FaultException class

Example 46 shows the definition of the `IT_Bus::FaultException` class. This is the class you must catch to handle an Artix system exception. Accessor and modifier functions are provided for all of the `FaultException` attributes.

Example 46: *The FaultException Class*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API FaultException :
        public SequenceComplexType,
        public Exception,
        public Rethrowable<FaultException>
    {
        ...
      public:
        FaultException(
            const FaultCategory::Category category,
            const String & namespace_uri,
            const String & code
        );

        FaultException();
        ...
        const FaultCategory & get_category() const;
        FaultCategory & get_category();
        void set_category(const FaultCategory & val);

        const String & get_namespace_uri() const;
        String & get_namespace_uri();
        void set_namespace_uri(const String & val);

        const String & get_code() const;
        String & get_code();
        void set_code(const String & val);

        const String & get_detail() const;
        String & get_detail();
        void set_detail(const String & val);

        const FaultSource & get_source() const;
```

Example 46: *The FaultException Class*

```
        FaultSource & get_source();
        void set_source(const FaultSource & val);

        const FaultCompletionStatus & get_completion_status()
                                                         const;
        FaultCompletionStatus & get_completion_status();
        void set_completion_status(
            const FaultCompletionStatus & val
        );

        const String & get_description() const;
        String & get_description();
        void set_description(const String & val);

        const String & get_server_id() const;
        String & get_server_id();
        void set_server_id(const String & val);
        ...
      private:
        ...
    };
}
```

IT_Bus::FaultCategory class

Example 47 shows the definition of the `IT_Bus::FaultCategory` class. This class provides the functions, `get_value()` and `set_value()`, to access or modify the fault category.

Example 47: *The FaultCategory Class*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API FaultCategory : public AnySimpleType
    {
      public:
        enum Category
        {
            NO_PERMISSION,
            CONNECTION_FAILURE,
            MARSHAL_ERROR,
            NOT_EXIST,
            TRANSIENT,
            UNKNOWN,
            TIMEOUT,
            VERSION_ERROR,
            NOT_UNDERSTOOD,
            MEMORY,
            BAD_OPERATION,
            INTERNAL,
            INVALID_REFERENCE,
            NOT_IMPLEMENTED,
            LICENSE
        };
        ...
```

Example 47: *The FaultCategory Class*

```
        FaultCategory();
        FaultCategory(const Category value);
        ...
        void set_value(const Category value);
        Category get_value() const;
        ...
    };
};
```

IT_Bus::FaultSource class

Example 48 shows the definition of the `IT_Bus::FaultSource` class. This class provides the functions, `get_value()` and `set_value()`, to access or modify the fault source.

Example 48: *The FaultSource Class*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API FaultSource : public AnySimpleType
    {
      public:
        enum Source
        {
            CLIENT,
            SERVER,
            UNKNOWN
        };
        ...
        FaultSource();
        FaultSource(const Source value);
        ...
        void set_value(const Source value);
        Source get_value() const;
        ...
    };
};
```

IT_Bus::FaultCompletionStatus class

Example 49 shows the definition of the `IT_Bus::FaultCompletionStatus` class. This class provides the functions, `get_value()` and `set_value()`, to access or modify the fault completion status.

Example 49: *The FaultCompletionStatus Class*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API FaultCompletionStatus : public AnySimpleType
    {
      public:
```

Example 49: *The FaultCompletionStatus Class*

```
        enum CompletionStatus
        {
            YES,
            NO,
            MAYBE
        };
        ...
        FaultCompletionStatus();
        FaultCompletionStatus(const CompletionStatus value);
        ...
        void set_value(const CompletionStatus value);
        CompletionStatus get_value() const;
        ...
    };
};
```

User-Defined Exceptions

Artix supports user-defined exceptions, which propagate from one Artix application to another. To define a user exception, you must declare the exception as a *fault* in WSDL. The WSDL-to-C++ compiler then generates the stub code that you need to raise and catch the exception.

FaultException class

User exceptions are derived from the IT_Bus::UserFaultException class, which is defined in <it_bus/user_fault_exception.h>. The IT_Bus::UserFaultException class extends IT_Bus::Exception.

Declaring a fault in WSDL

Example 50 shows an example of a WSDL fault which can be raised on the echoInteger operation. The format of the fault message is specified by the tns:SampleFault message.

Example 50: *Declaration of the faultMessage Fault*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="http://www.iona.com/userfault"
    xmlns="http://schemas.xmlsoap.org/wsdl/"  ... >
    <types>
       <schema targetNamespace="http://www.iona.com/userfault"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
           <element name="my_exceptionElement"
                   type="tns:my_exceptionType"/>
1          <complexType name="my_exceptionType">
               <sequence>
                   <element name="ErrorMsg" type="xsd:string"/>
                   <element name="ErrorID" type="xsd:int"/>
               </sequence>
           </complexType>
        </schema>
```

Example 50: *Declaration of the faultMessage Fault*

```
    </types>
    <message name="requestMessage"/>
    <message name="responseMessage"/>
2   <message name="faultMessage">
        <part element="tns:my_exceptionElement"
              name="my_exceptionDetails"/>
    </message>

    <portType name="Receiver">
        <operation name="pingMe">
            <input  message="tns:requestMessage"
                    name="pingMeRequest"/>
            <output message="tns:responseMessage"
                    name="pingMeResponse"/>
3           <fault message="tns:faultMessage"
                   name="pingMeFault"/>
        </operation>
    </portType>
    ...
</definitions>
```

The preceding WSDL extract can be explained as follows:

1. If the fault is to hold more than one piece of data, you must declare a complex type for the fault data (in this case, my_exceptionType holds an error message string, ErrorMsg, and an error ID, ErrorID).
2. Declare a message for the fault, containing just a single part. The WSDL specification allows only single-part messages in a fault—multi-part messages are *not* allowed.
3. The <fault> tag must be added to the scope of the operation (or operations) which can raise this particular type of fault.

> **Note:** There is no limit to the number of <fault> tags that can be included in an operation element.

C++ mapping of user fault

When the user fault is mapped to C++, two classes are generated to represent the exception.

The first class, faultMessageException, represents the fault message, faultMessage. This class, which inherits from IT_Bus::UserFaultException, is the class that you actually throw and catch as an exception in C++. Example 51 shows the definition of the faultMessageException class.

Example 51: *The faultMessageException Class*

```
// C++
namespace userfault
{
    class faultMessageException
    : public IT_Bus::UserFaultException,
      public
   IT_Bus::Rethrowable<userfault::faultMessageException>
```

Example 51: *The faultMessageException Class*

```
    {
      public:
        ...
        faultMessageException();
        ...
        virtual const IT_Bus::QName &
        get_message_name() const;

        my_exceptionType &       getmy_exceptionDetails();
        const my_exceptionType & getmy_exceptionDetails() const;
        void setmy_exceptionDetails(const my_exceptionType & val);

      private:
        ...
    };
};
```

The `get_message_name()` function returns the name of the user exception. The `faultMessageException` class declares functions, `get`*PartName*`()` and `set`*PartName*`()`, for accessing and modifying the message part (there is only one part in the message). For example, the `getmy_exceptionDetails()` function returns a reference to a `my_exceptionType` object.

The second class, `my_exceptionType`, represents the exception data. Example 52 shows the definition of the `my_exceptionType` class. This class provides accessor and modifier functions for the `ErrorMsg` and `ErrorID` exception members.

Example 52: *The my_exceptionType Class*

```
// C++
...
namespace userfault
{
    ...
    class my_exceptionType : public IT_Bus::SequenceComplexType
    {
      public:
        ...
        my_exceptionType();
        ...
        IT_Bus::String &       getErrorMsg();
        const IT_Bus::String & getErrorMsg() const;
        void setErrorMsg(const IT_Bus::String & val);

        IT_Bus::Int       getErrorID();
        const IT_Bus::Int getErrorID() const;
        void setErrorID(const IT_Bus::Int val);

      private:
        ...
    };
};
```

Raising a fault exception in a server

Example 53 shows how to raise the `faultMessageException` exception in the server code. This implementation of `pingMe` always throws the user exception, `faultMessageException`.

Example 53: *Raising a faultMessageException in the Server*

```
// C++
void
ReceiverImpl::pingMe() IT_THROW_DECL((IT_Bus::Exception))
{
  // Initialize an instance of the my_exceptionType
  my_exceptionType exception_details;

  // Set ErrorMsg and ErrorID
  exception_details.setErrorMsg("pingMe: No implementation");
  exception_details.setErrorID(555);

  // Now set exception details into faultMessageException
  faultMessageException the_exception;
  the_exception.setmy_exceptionDetails(exception_details);

  // Throw the exception
  throw the_exception;
}
```

Catching a fault exception in a client

Example 54 shows how to catch the `faultMessageException` exception on the client side. The client uses the proxy instance, `client`, to call the `pingMe` operation remotely.

Example 54: *Catching faultMessageException in the Client*

```
// C++

// Create an instance of the web service client
IT_Bus::init(argc, argv);

try
{
    ReceiverClient client;

    client.pingMe ();
}
catch (const faultMessageException& ex)
{
    my_exceptionType exception_details
                          = ex.getmy_exceptionDetails();

    // Now display the details of the exception
    cout << "Exception Message: "
         << exception_details.getErrorMsg() << endl;
    cout << "Exception ID: "
         << exception_details.getErrorID() << endl;
}
```

Memory Management

This section discusses the memory management rules for Artix types, particularly for generated complex types.

Managing Parameters

This subsection discusses the guidelines for managing the memory for parameters of complex type. In Artix, memory management of parameters is relatively straightforward, because the Artix C++ mapping passes parameters by reference.

Note: If you use pointer types to reference operation parameters, see “Smart Pointers” on page 100 for advice on memory management.

Memory management rules

There are just two important memory management rules to remember when writing an Artix client or server:

1. The client is responsible for deallocating parameters.
2. If the server needs to keep a copy of parameter data, it must make a copy of the parameter. In general, parameters are deallocated as soon as an operation returns.

WSDL example

Example 55 shows an example of a WSDL operation, `testSeqParams`, with three parameters, `inSeq`, `inoutSeq`, and `outSeq`, of sequence type, `xsd1:SequenceType`.

Example 55: *WSDL Example with in, inout and out Parameters*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >
    <types>
        <schema targetNamespace="http://soapinterop.org/xsd"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
          <complexType name="SequenceType">
              <sequence>
                  <element name="varFloat" type="xsd:float"/>
                  <element name="varInt" type="xsd:int"/>
                  <element name="varString" type="xsd:string"/>
              </sequence>
          </complexType>
            ...
       </schema>
    </types>
    ...
    <message name="testSeqParams">
        <part name="inSeq" type="xsd1:SequenceType"/>
        <part name="inoutSeq" type="xsd1:SequenceType"/>
    </message>
```

Example 55: *WSDL Example with in, inout and out Parameters*

```
    <message name="testSeqParamsResponse">
        <part name="inoutSeq" type="xsd1:SequenceType"/>
        <part name="outSeq" type="xsd1:SequenceType"/>
    </message>
    ...
    <portType name="BasePortType">
        <operation name="testSeqParams">
            <input message="tns:testSeqParams"
                   name="testSeqParams"/>
            <output message="tns:testSeqParamsResponse"
                    name="testSeqParamsResponse"/>
        </operation>
    ...
    </portType>
    ...
</definitions>
```

Client example

Example 56 shows how to allocate, initialize, and deallocate parameters when calling the `testSeqParams` operation.

Example 56: *Client Calling the testSeqParams Operation*

```
// C++
try
{
    IT_Bus::init(argc, argv);

1   BaseClient bc;

2   // Allocate all parameters
    SequenceType inSeq, inoutSeq, outSeq;

3   // Initialize in and inout parameters
    inSeq.setvarFloat((IT_Bus::Float) 1.234);
    inSeq.setvarInt(54321);
    inSeq.setvarString("One, two, three");
    inoutSeq.setvarFloat((IT_Bus::Float) 4.321);
    inoutSeq.setvarInt(12345);
    inoutSeq.setvarString("Four, five, six");

    // Call the 'testSeqParams' operation
    bc.testSeqParams(inSeq, inoutSeq, outSeq);

4   // End of scope:
    // Implicit deallocation of inSeq, inoutSeq, and outSeq.
}
catch(IT_Bus::Exception& e)
{
    cout << endl << "Caught Unexpected Exception: "
         << endl << e.message()
         << endl;
    return -1;
}
```

The preceding client example can be explained as follows:

1. This line creates an instance of the client proxy, `bc`, which is used to invoke the WSDL operations.
2. You must allocate memory for *all* kinds of parameter, in, inout, and out. In this example, the parameters are created on the stack.
3. You initialize *only* the in and inout parameters. The server will initialize the out parameters.
4. It is the responsibility of the client to deallocate all kinds of parameter. In this example, the parameters are all deallocated at the end of the current scope, because they have been allocated on the stack.

Server example

Example 57 shows how the parameters are used on the server side, in the C++ implementation of the `testSeqParams` operation.

Example 57: *Server Calling the testSeqParams Operation*

```
// C++
void
BaseImpl::testSeqParams(
    const SequenceType & inSeq,
    SequenceType & inoutSeq,
    SequenceType & outSeq
) IT_THROW_DECL((IT_Bus::Exception))
{
    cout << "BaseImpl::testSeqParams called" << endl;

1   // Print inSeq
    cout << "inSeq.varFloat  = " << inSeq.getvarFloat() << endl;
    cout << "inSeq.varInt    = " << inSeq.getvarInt() << endl;
    cout << "inSeq.varString = " << inSeq.getvarString() <<
   endl;

2   // (Optionally) Copy in/inout parameters
    // ...

3   // Print and change inoutSeq
    cout << "inoutSeq.varFloat  = "
         << inoutSeq.getvarFloat() << endl;
    cout << "inoutSeq.varInt    = "
         << inoutSeq.getvarInt() << endl;
    cout << "inoutSeq.varString = "
         << inoutSeq.getvarString() << endl;
    inoutSeq.setvarFloat(2.0);
    inoutSeq.setvarInt(2);
    inoutSeq.setvarString("Two");

4   // Initialize outSeq
    outSeq.setvarFloat(3.0);
    outSeq.setvarInt(3);
    outSeq.setvarString("Three");
}
```

The preceding server example can be explained as follows:

1. The server programmer has read-only access to the in parameters (they are declared `const` in the operation signature).
2. If you want to access data from in or inout parameters after the operation returns, you must copy them (deep copy). It would be an error to use the & operator to obtain a pointer to the parameter data, because the Artix server stub deallocates the parameters as soon as the operation returns.

   See “Assignment and Copying” on page 99 for details of how to copy Artix data types.
3. You have read/write access to the inout parameters.
4. You should initialize each of the out parameters (otherwise they will be returned with default initial values).

Assignment and Copying

The WSDL-to-C++ compiler generates copy constructors and assignment operators for all complex types.

Copy constructor

The WSDL-to-C++ compiler generates a copy constructor for complex types. For example, the `SequenceType` type declared in Example 55 on page 96 has the following copy constructor:

```
// C++
SequenceType(const SequenceType& copy);
```

This enables you to initialize `SequenceType` data as follows:

```
// C++
SequenceType original;
original.setvarFloat(1.23);
original.setvarInt(321);
original.setvarString("One, two, three.");

SequenceType copy_1(original);
SequenceType copy_2 = original;
```

Assignment operator

The WSDL-to-C++ compiler generates an assignment operator for complex types. For example, the generated assignment operator enables you to assign a `SequenceType` instance as follows:

```
// C++
SequenceType original;
original.setvarFloat(1.23);
original.setvarInt(321);
original.setvarString("One, two, three.");

SequenceType assign_to;

assign_to = original;
```

Recursive copying

In WSDL, complex types can be nested inside each other to an arbitrary degree. When such a nested complex type is mapped to C++ by Artix, the copy constructor and assignment operators are designed to copy the nested members recursively (deep copy).

Deallocating

Using delete

In C++, if you allocate a complex type on the heap (that is, using pointers and `new`), you can generally delete the data instance using the `delete` operator. It is usually better, however, to use smart pointers in this context—see "Smart Pointers" on page 100.

Recursive deallocation

The Artix C++ types are designed to support recursive deallocation.

That is, if you have an instance, `T`, of a complex type which has other complex types nested inside it, the entire memory for the complex type including its nested members would be deallocated when you delete `T`. This works for complex types nested to an arbitrary degree.

Smart Pointers

To help you avoid memory leaks when using pointers, the WSDL-to-C++ compiler generates a smart pointer class, *ComplexType*`Ptr`, for every generated complex type, *ComplexType*. The following aspects of smart pointers are discussed here:

- What is a smart pointer?
- Artix smart pointers.
- Client example using simple pointers.
- Client example using smart pointers.

What is a smart pointer?

A smart pointer class is a C++ class that overloads the `*` (dereferencing) and `->` (member access) operators, in order to imitate the syntax of an ordinary C++ pointer.

Artix smart pointers

Artix smart pointers are defined using a template class, IT_AutoPtr<T>, which has the same API as the standard auto pointer template, auto_ptr<T>, from the C++ standard template library. If the standard library is supported on the platform, IT_AutoPtr is simply a typedef of std::auto_ptr.

For example, the SequenceTypePtr smart pointer class is defined by the following generated typedef:

```
// C++
typedef IT_AutoPtr<SequenceType> SequenceTypePtr;
```

The key feature that makes this pointer type smart is that the destructor always deletes the memory the pointer is pointing at. This feature ensures that you cannot leak memory when it is referenced by a smart pointer.

Client example using simple pointers

Example 58 shows how to call the testSeqParams operation using parameters that are allocated on the heap and referenced by *simple pointers*

Example 58: *Client Calling testSeqParams Using Simple Pointers*

```
// C++
try
{
    IT_Bus::init(argc, argv);

    BaseClient bc;

1   // Allocate all parameters
    SequenceType *inSeqP    = new SequenceType();
    SequenceType *inoutSeqP = new SequenceType();
    SequenceType *outSeqP   = new SequenceType();

    // Initialize in and inout parameters
    inSeqP->setvarFloat((IT_Bus::Float) 1.234);
    inSeqP->setvarInt(54321);
    inSeqP->setvarString("One, two, three");
    inoutSeqP->setvarFloat((IT_Bus::Float) 4.321);
    inoutSeqP->setvarInt(12345);
    inoutSeqP->setvarString("Four, five, six");

    // Call the 'testSeqParams' operation
    bc.testSeqParams(*inSeqP, *inoutSeqP, *outSeqP);

2   // End of scope:
    delete inSeqP;
    delete inoutSeqP;
    delete outSeqP;
}
catch(IT_Bus::Exception& e)
{
    cout << endl << "Caught Unexpected Exception: "
```

Example 58: *Client Calling testSeqParams Using Simple Pointers*

```
         << endl << e.message()
         << endl;
    return -1;
}
```

The preceding client example can be explained as follows:

1. The parameters are allocated on the heap.
2. Before you reach the end of the current scope, you *must* explicitly delete the parameters or the memory will be leaked.

Client example using smart pointers

Example 59 shows how to call the `testSeqParams` operation using parameters that are allocated on the heap and referenced by *smart pointers*

Example 59: *Client Calling testSeqParams Using Smart Pointers*

```
// C++
try
{
    IT_Bus::init(argc, argv);

    BaseClient bc;

    // Allocate all parameters
1   SequenceTypePtr inSeqP(new SequenceType());
    SequenceTypePtr inoutSeqP(new SequenceType());
    SequenceTypePtr outSeqP(new SequenceType());

    // Initialize in and inout parameters
    inSeqP->setvarFloat((IT_Bus::Float) 1.234);
    inSeqP->setvarInt(54321);
    inSeqP->setvarString("One, two, three");
    inoutSeqP->setvarFloat((IT_Bus::Float) 4.321);
    inoutSeqP->setvarInt(12345);
    inoutSeqP->setvarString("Four, five, six");

    // Call the 'testSeqParams' operation
    bc.testSeqParams(*inSeqP, *inoutSeqP, *outSeqP);

2   // End of scope:
    // Parameter data automatically deallocated by smart pointers
}
catch(IT_Bus::Exception& e)
{
    cout << endl << "Caught Unexpected Exception: "
         << endl << e.message()
         << endl;
    return -1;
}
```

The preceding client example can be explained as follows:

1. The parameters are allocated on the heap, using smart pointers of `SequenceTypePtr` type.
2. In this case, there is no need to deallocate the parameter data explicitly. The smart pointers, `inSeqP`, `inoutSeqP`, and `outSeqP`, automatically delete the memory they are pointing at when they go out of scope.

Multi-Threading

This section provides an overview of threading in Artix and describes the issues affecting multi-threaded clients and servers in Artix.

Client Threading Issues

Client threading

The runtime library is thread-safe, in that multi-threaded applications may safely use the library from multiple threads simultaneously.

Moreover, the client stub code is thread-safe by default. That is, you can safely share a single proxy instance amongst multiple threads. The Artix stub code uses mutex locks to protect the proxy instance from concurrent access by multiple threads.

> **Note:** Versions of Artix prior to 4.0 are *not* thread-safe by default. In these older Artix versions, it was possible to enable thread-safe proxies by calling the `IT_Bus::Port::set_threading_model()` function. For backward compatibility reasons, the `set_threading_model()` function is still available in Artix 4.0, but it has no effect.

Servant Threading Models

Artix supports a variety of different threading models on the server side. The threading model that applies to a particular service can be specified by programming (see "Setting the Servant Threading Model" on page 106). This subsection provides an overview of each of the servant threading models in Artix, as follows:

- Multi-threaded.
- Serialized.
- Per-port.
- PerThread.
- PerInvocation.

Default threading model

The default threading model is multi-threaded.

Multi-threaded

The *multi-threaded* threading model implies that a single instance is created and shared on multiple threads. The servant object must expect to be called from multiple threads simultaneously.

Figure 11 shows an outline of the multi-threaded threading model. In this case, the threads all share the same servant instance.

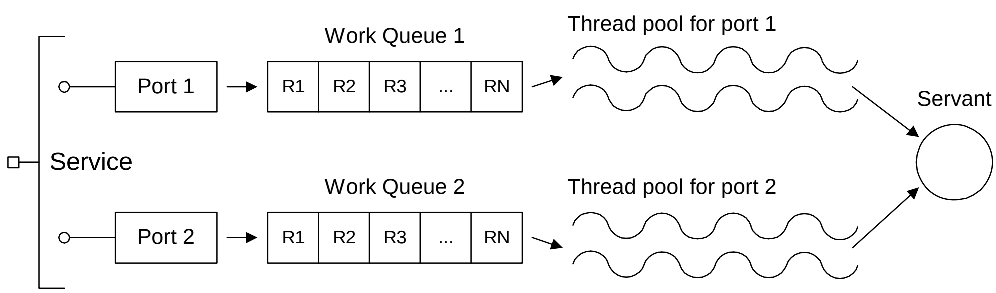

Figure 11: *Outline of the Multi-Threaded Threading Model*

Serialized

The `Serialized` threading model implies that access to the servant is serialized (implemented using mutex locks). The servant object can be called from no more than one thread at a time.

Figure 12 shows an outline of the `Serialized` threading model. In this case, the threads all share the same servant instance, but access is serialized.

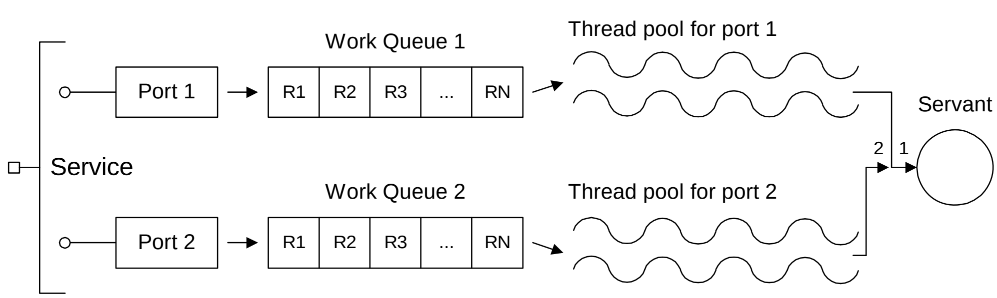

Figure 12: *Outline of the Serialized Threading Model*

Per-port

The *per-port* threading model implies that a servant instance is created per port. Each servant object must expect to be called from multiple threads simultaneously, because each port has an associated thread pool.

Figure 13 shows an outline of the `PerPort` threading model. In this case, the threads in a thread pool share the same servant instance.

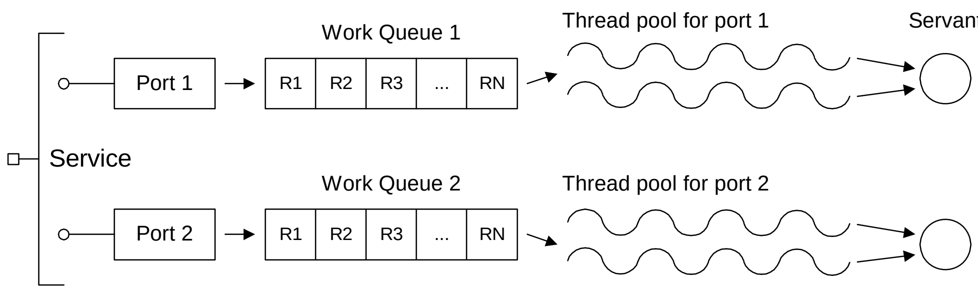

Figure 13: *Outline of the Per-Port Threading Model*

PerThread

The `PerThread` threading model implies that a servant instance is created per thread. This allows the servant objects to use thread-local storage, resources with thread affinity (like MQ), and reduces synchronization overhead.

Figure 14 shows an outline of the `PerThread` threading model. An Artix service can have multiple ports, and each of the ports is served by a work queue that stores the incoming requests. A pool of threads is reserved for each port, and each thread in the pool is associated with a distinct servant instance.

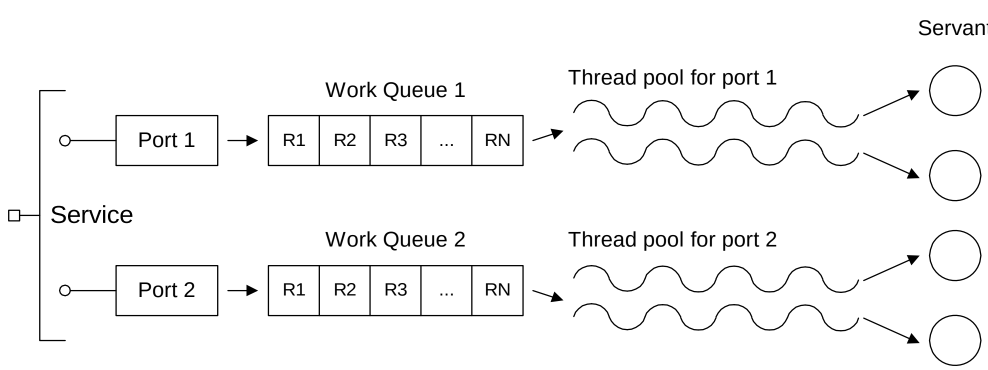

Figure 14: *Outline of the PerThread Threading Model*

PerInvocation

The PerInvocation threading model implies that a servant instance is created for every invocation. In this case, the servant implementation does not need to be thread-safe, because a servant can be called from no more than one thread at a time.

The relationship between threads and servants is similar to the case of the PerThread threading model (see Figure 14 on page 105). There is a difference in servant lifecycle management, however. Each thread is associated with a servant for the duration of an operation invocation. At the end of the invocation, the servant instance is destroyed.

Setting the Servant Threading Model

Some of the servant threading models are implemented using *wrapper servant* classes, which work by overriding the default behavior of a servant's dispatch() function. Exceptions to this pattern are the default multi-threaded model and the per-port threading model. This section describes how to program the various servant threading models.

How to set a per-port threading model

The per-port threading model can be enabled by employing the two-step style of servant registration (see "Activating a static servant" on page 54 or "Activating a transient servant" on page 59). For example, you could register distinct servants, corba_servant and soap_servant, against distinct ports, CORBAPort and SOAPPort, using the following code example:

```
// C++
IT_Bus::QName service_name("", "BankService",
   "http://www.iona.com/bus/demos/bank");

IT_Bus::Service_var bank_service =
    bus->add_service("bank.wsdl", service_name);
bank_service->register_servant(corba_servant, "CORBAPort");
bank_service->register_servant(soap_servant, "SOAPPort");
```

Wrapper servants

The only wrapper servant function that you need is a constructor. Example 60 shows the constructors for each of the wrapper servant classes.

Example 60: *Constructors for the Wrapper Servant Classes*

```
// C++
IT_Bus::SerializedServant(IT_Bus::Servant& servant);

IT_Bus::PerThreadServant(IT_Bus::Servant& servant);

IT_Bus::PerInvocationServant(IT_Bus::Servant& servant);
```

How to set a threading model using wrapper servants

To register a servant with a `Serialized`, `PerThread` or `PerInvocation` threading model, perform the following steps:

- Step 1—Implement the servant clone() function (if required).
- Step 2—Register the wrapper servant.

Step 1—Implement the servant clone() function (if required)

If you intend to use a `PerThread` or `PerInvocation` threading model, you must implement the `clone()` function in your servant class. The `clone()` function will be called automatically whenever the threading model demands a new servant instance. Example 61 shows the default implementation of the `clone()` function for the servant class, *PortType*`Impl`.

Example 61: *Default Implementation of the clone() Function*

```
// C++
IT_Bus::Servant*
PortTypeImpl::clone() const
{
    return new PortTypeImpl(get_bus());
}
```

Step 2—Register the wrapper servant

To register a wrapper servant, you must pass the original servant object to a wrapper servant constructor and then pass the wrapper servant to the `register_servant()` function (or the `register_transient_servant()` function in the case of transient servants).

For example, Example 62 shows how the main function of the bank server example can be modified to register the `BankImpl` servant with a `PerThread` threading model.

Example 62: *Registering a Servant with a PerThread Threading Model*

```
// C++
...
try {
   IT_Bus::Bus_var bus = IT_Bus::init(argc, (char **)argv);

   BankImpl my_bank(bus);
1  IT_Bus::PerThreadServant per_thread_bank(my_bank);

   QName service_name("", "BankService", "http://www.iona.com/bus/demos/bank");
```

Example 62: *Registering a Servant with a PerThread Threading Model*

2
```
    bus->register_servant(
        per_thread_bank,
        "../wsdl/bank.wsdl",
        service_name
    );

    IT_Bus::run();

    bus->remove_service(service_name);
}
catch (IT_Bus::Exception& e) { ... }
```

The preceding C++ code can be described as follows:

1. In this step, the `BankImpl` servant is wrapped by a new `IT_Bus::PerThreadServant` instance.
2. When it comes to registering, you must register the *wrapper servant*, `per_thread_bank`, instead of the original servant, `my_bank`.

Thread Pool Configuration

Thread pool settings

The thread pool for each port is controlled by the following parameters (which can be set in the configuration):

- *Initial threads*—the number of threads initially created for each port.
- *Low water mark*—the size of the dynamically allocated pool of threads will not fall below this level.
- *High water mark*—the size of the dynamically allocated pool of threads will not rise above this level.

Thread pools are configured by adding to or editing the settings in the *ArtixInstallDir*`/etc/domains/artix.cfg` configuration file. In the following examples, it is assumed that the Artix application specifies its configuration scope to be `sample_config`.

Note: You can specify the configuration scope at the command line by passing the switch `-BUSname` *ConfigScopeName* to the Artix executable. Command-line arguments are normally passed to `IT_Bus::init()`.

Thread pool configuration levels

Thread pools can be configured at several levels, where the more specific configuration settings take precedence over the less specific, as follows:

- Global level.
- Service name level.
- Qualified service name level.

Global level

The variables shown in Example 63 can be used to configure thread pools at the global level; that is, these settings would apply to all services by default.

Example 63: *Thread Pool Settings at the Global Level*

```
# Artix configuration file

sample_config {
    ...
    # Thread pool settings at global level
    thread_pool:initial_threads = "3";
    thread_pool:low_water_mark  = "5";
    thread_pool:high_water_mark = "10";
};
```

The default settings are as follows:

```
thread_pool:initial_threads = "2";
thread_pool:low_water_mark  = "5";
thread_pool:high_water_mark = "25";
```

Service name level

To configure thread pools at the service name level (that is, overriding the global settings for a specific service only), set the following configuration variables:

`thread_pool:initial_threads:`*ServiceName*
`thread_pool:low_water_mark:`*ServiceName*
`thread_pool:high_water_mark:`*ServiceName*

Where *ServiceName* is the name of the particular service to configure, as it appears in the WSDL `<service name="`*ServiceName*`">` tag.

For example, the settings in Example 64 show how to configure the thread pool for a service named `SessionManager`.

Example 64: *Thread Pool Settings at the Service Name Level*

```
# Artix configuration file

sample_config {
    ...
    # Thread pool settings at Service name level
    thread_pool:initial_threads:SessionManager = "1";
    thread_pool:low_water_mark:SessionManager  = "5";
    thread_pool:high_water_mark:SessionManager = "10";
};
```

Qualified service name level

Occasionally, if the service names from two different namespaces clash, it might be necessary to identify a service by its fully-qualified service name. To configure thread pools at the qualified service name level, set the following configuration variables:

thread_pool:initial_threads:*NamespaceURI*:*ServiceName*
thread_pool:low_water_mark:*NamespaceURI*:*ServiceName*
thread_pool:high_water_mark:*NamespaceURI*:*ServiceName*

Where *NamespaceURI* is the namespace URI in which *ServiceName* is defined.

For example, the settings in Example 65 show how to configure the thread pool for a service named `SessionManager` in the `http://my.tns1/` namespace URI.

Example 65: *Thread Pool Settings at the Qualified Service Name Level*

```
# Artix configuration file

sample_config {
    ...
    # Thread pool settings at Service name level
    thread_pool:initial_threads:http://my.tns1/:SessionManager = "1";
    thread_pool:low_water_mark:http://my.tns1/:SessionManager = "5";
    thread_pool:high_water_mark:http://my.tns1/:SessionManager = "10";
};
```

Converting with to_string() and from_string()

This section describes how you can use the `<<` operator, the `IT_Bus::to_string()` function and the `IT_Bus::from_string()` function to convert Artix data types to and from a string format.

Header files

The following header files must be included in your source code to access the string conversion APIs:

- <it_bus/to_string.h>
- <it_bus/from_string.h>

Library

To use the string conversion functions and operators, link your application with the following library:

- `it_bus_xml.lib`, on Windows platforms,
- `libit_bus_xml[.a][.so]`, on UNIX platforms.

Demonstration

The following demonstration gives an example of how to use the Artix string conversion functions, `to_string()` and `from_string()`:

ArtixInstallDir/cxx_java/samples/basic/to_string

Example struct

Example 66 shows the definition of an XML schema type, `SimpleStruct`, which is used by the string conversion examples in this section.

Example 66: *Schema Definition of a SimpleStruct Type*

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://schemas.iona.com/tests/type_test"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://schemas.iona.com/tests/type_test">

    <complexType name="SimpleStruct">
        <sequence>
            <element name="varFloat" type="float"/>
            <element name="varInt" type="int"/>
            <element name="varString" type="string"/>
        </sequence>
        <attribute name="varAttrString" type="string"/>
    </complexType>

</schema>
```

operator<<()

By including the `<it_bus/to_string.h>` header file and linking with the `it_bus_xml` library, you can use the `<<` operator to print out any Artix data type in a string format (assuming that the stub code for this data type is already linked with your application).

Example using <<

The following code example shows how to print a simple struct, `first_struct`, as a string using the `<<` stream operator:

```
// C++
...
#include <it_bus/to_string.h>
...
int main(int argc, char** argv)
{
    SimpleStruct first_struct;
    first_struct.setvarString("goodbye");
    first_struct.setvarInt(121);
    first_struct.setvarFloat(3.14);

    cout << endl << "Print using operator<<"
```

```
        << endl << first_struct << endl;
}
```

The preceding code produces the following output:

```
Print using operator<<
<?xml version='1.0' encoding='utf-8'?><to_string
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <varFloat>3.140000105e0</varFloat><varInt>121</varInt>
   <varString>goodbye</varString></to_string>
```

In the stringified output, the element name defaults to <to_string>.

to_string()

Example 67 shows the signature of the `IT_Bus::to_string()` function, as defined in the <it_bus/to_string.h> header.

Example 67: *Signature of the IT_Bus::to_string() Function*

```
// C++
namespace IT_Bus
{
    String IT_BUS_XML_API
    to_string(
        const AnyType& data,
       const QName& element_name=default_to_string_element_name
    );
}
```

You can convert any Artix data type to a string, `IT_Bus::String`, by passing it as the first argument in `to_string()` (`IT_Bus::AnyType` is the base class for all Artix data types). The resulting string has the following general format:

```
<?xml version='1.0' encoding='utf-8'?>
<ElementName
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    ...
</ElementName>
```

Where the *ElementName* has one of the following values:

- If the second parameter of `to_string()` is defaulted, the *ElementName* is `to_string`.
- If the second parameter of `to_string()` is a simple string, say `foo`, the ElementName is `foo`.
- If the second parameter of `to_string()` is an `IT_Bus::QName`, say `QName("", "foo", "http://xml.iona.com/IDD/test")`, the *ElementName* is `m1:foo`, where `m1` is the prefix associated with the `http://xml.iona.com/IDD/test` namespace URI.

Example using to_string()

The following code example shows how to convert a simple struct, `second_struct`, to a string using the `to_string()` function:

```
// C++
...
#include <it_bus/to_string.h>
...
int main(int argc, char** argv)
{
    SimpleStruct first_struct;
    second_struct.setvarString("hello");
    second_struct.setvarInt(2);
    second_struct.setvarFloat(1.1);

    String resulting_xml = IT_Bus::to_string(
        second_struct,
        QName("", "foo", "http://xml.iona.com/IDD/test")
    );

    cout << endl << "Resulting XML String:"
         << endl << resulting_xml.c_str() << endl;
}
```

The preceding code produces the following output:

```
Resulting XML String:
<?xml version='1.0' encoding='utf-8'?><m1:foo
   xmlns:m1="http://xml.iona.com/IDD/test"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <varFloat>1.100000024e0</varFloat><varInt>2</varInt>
   <varString>hello</varString></m1:foo>
```

In the stringified output, the element name is defined as `m1:foo`.

from_string()

Example 68 shows the signature of the `IT_Bus::from_string()` function, as defined in the `<it_bus/from_string.h>` header.

Example 68: *Signature of the IT_Bus::from_string() Function*

```
// C++
namespace IT_Bus
{
    void IT_BUS_XML_API
    from_string(
        const String & data,
        AnyType & result,
        const QName &
            element_name=default_from_string_element_name
    );
}
```

You can initialize an Artix data type from an XML element in string format using the `from_string()` conversion function. Pass the XML string as the first argument, `data`, and the data type to initialize as the second parameter, `result`.

Example using from_string()

The following code example shows how to convert an XML string, `original_xml`, to a simple struct, `simple_struct`, using the `from_string()` function:

```
// C++
...
#include <it_bus/from_string.h>
...
int main(int argc, char** argv)
{
   String original_xml = "<?xml version='1.0' encoding='utf-8'?>
   <to_string xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
   xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"><varFloat>1.100000024e0</varFloat>
   <varInt>2</varInt><varString>hello</varString></to_string>";

    SimpleStruct simple_struct;

    IT_Bus::from_string(original_xml, simple_struct);

    cout << endl << "Output values of SimpleStruct C++ type using accessor methods."
         << endl << "    SimpleStruct populated with the following values:"
        << endl << "    SimpleStruct::varString = " << simple_struct.getvarString().c_str()
         << endl << "    SimpleStruct::varInt = " << simple_struct.getvarInt()
         << endl << "    SimpleStruct::varFloat = " << simple_struct.getvarFloat() << endl;
}
```

Locating Services with UDDI

A Universal Description, Discovery and Integration (UDDI) registry is a form of database that enables you to store and retrieve Web services endpoints. It is particularly useful as a means of making Web services available on the Internet. Instead of making your WSDL contract available to clients in the form of a file, you can publish the WSDL contract to a UDDI registry. Clients can then query the UDDI registry and retrieve the WSDL contract at runtime.

Publishing WSDL to UDDI

You can publish your WSDL contract either to a local UDDI registry or to a public UDDI registry, such as `http://uddi.ibm.com` from IBM or `http://uddi.microsoft.com/` from Microsoft. To publish your WSDL contract, navigate to one of the public UDDI Web sites and follow the instructions there.

A list of public UDDI registries is available from WSINDEX (http://www.wsindex.org/UDDI/Registries/index.html).

UDDI URL format

Artix uses UDDI query strings that take the form of a URL:

uddi:<*UDDIRegistryEndpointURL*>?<*QueryString*>

The UDDI URL is built up from the following components:

- *UDDIRegistryEndpointURL*—the endpoint address of a UDDI registry. This could either be a local UDDI registry (for example, http://localhost:9000/services/uddi/inquiry) or a public UDDI registry on the Internet (for example, http://uddi.ibm.com/ubr/inquiryapi for IBM's UDDI registry).
- *QueryString*—a combination of attributes that is used to query the UDDI database for the Web service endpoint data. Currently, Artix only supports the tmodelname attribute. An example of a query string is:

  ```
  tmodelname=helloworld
  ```

  Within a query component, the characters ;, /, ?, :, @, &, =, +, ,, and $ are reserved.

Examples of valid UDDI URLs

```
uddi:http://localhost:9000/services/uddi/inquiry?tmodelname=hell
   oworld
uddi:http://uddi.ibm.com/ubr/inquiryapi?tmodelname=helloworld
```

Initializing a client proxy with UDDI

To initialize a client proxy with UDDI, pass a valid UDDI URL string to the proxy constructor. For example, if you have a local UDDI registry, http://localhost:9000/services/uddi/inquiry, where you have registered the WSDL contract from the HelloWorld demonstration (this contract is in *InstallDir*/cxx_java/samples/basic/hello_world_soap_http/etc), you can initialize the GreeterClient proxy as follows:

```
// C++
...
IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);

// Instantiate an instance of the proxy
GreeterClient hw("uddi:http://localhost:9000/services/uddi/inquiry?tmodelname=helloworld");

String string_out;

// Invoke sayHi operation
hw.sayHi(string_out);
```

Configuration

To configure an Artix client to support UDDI, you must add uddi_proxy to the application's orb_plugins list (for the C++ plug-in). For example:

```
# Artix Configuration File

my_application_scope {
    orb_plugins = [ ..., "uddi_proxy"];
    ...
};
```

Compiling and Linking an Artix Application

Compiler Requirements

An application built using Artix requires a number of -supplied C++ header files in order to compile. The directory containing these include files must be added to the include path for the compiler, so that when the compiler processes the generated files, it is able to find the necessary included infrastructure header files.

The following include path directives should be given to the compiler:

```
-I"$(IT_PRODUCT_DIR)\artix\$(IT_PRODUCT_VER)\include"
```

Linker Requirements

A number of Artix libraries are required to link with an application built using Artix. The following directives should be given to the linker:

```
-L"$(IT_PRODUCT_DIR)\artix\$(IT_PRODUCT_VER)\lib" it_bus.lib it_afc.lib it_art.lib it_ifc.lib
```

Table 1 shows the libraries that are required for linking an Artix application and their function.

Table 1: *Artix Import Libraries for Linking with an Application*

Windows Libraries	UNIX Libraries	Description
it_bus.lib	libit_bus.so libit_bus.sl libit_bus.a	The Bus library provides the functionality required to access the Artix bus. Required for all applications that use Artix functionality.
it_afc.lib	libit_afc.so libit_afc.sl libit_afc.a	The Artix foundation classes provide Artix specific data type extensions such as IT_Bus::Float, etc. Required for all applications that use Artix functionality.
it_ifc.lib	libit_ifc.so libit_ifc.sl libit_ifc.a	The Artix foundation classes provide proprietary data types and exceptions.

Table 1: *Artix Import Libraries for Linking with an Application*

Windows Libraries	UNIX Libraries	Description
it_art.lib	libit_art.so libit_art.sl libit_art.a	The ART library provides advanced programming functionality that requires access to the Artix infrastructure and the underlying ORB.

Runtime Requirements

The following directories need to be in the path, either by copying them into a location already in the path, or by adding their locations to the path. The following lists the required libraries and their location in the distribution files (all paths are relative to the root directory of the distribution):

```
"$(IT_PRODUCT_DIR)\bin"
```

On some UNIX platforms you also have to update the SHLIB_PATH or LD_LIBRARY_PATH variables to include the Artix shared library directory.

Building Artix Stub Libraries on Windows

The Artix WSDL-to-C++ compiler features an option, -declspec, that simplifies the process of building Dynamic Linking Libraries (DLLs) on the Windows platform. The -declspec option defines a macro that automatically inserts export declarations into the stub header files.

Generating stubs with declaration specifiers

To generate Artix stubs with declaration specifiers, use the -declspec option to the WSDL-to-C++ compiler, as follows:

```
wsdltocpp -declspec MY_DECL_SPEC BaseService.wsdl
```

In this example, the -declspec option would add the following preprocessor macro definition to the top of the generated header files:

```
#if !defined(MY_DECL_SPEC)
#if defined(MY_DECL_SPEC_EXPORT)
#define MY_DECL_SPEC     IT_DECLSPEC_EXPORT
#else
#define MY_DECL_SPEC     IT_DECLSPEC_IMPORT
#endif
#endif
```

Where the IT_DECLSPEC_EXPORT macro is defined as _declspec(dllexport) and the IT_DECLSPEC_IMPORT macro is _declspec(dllimport).

Each class in the header file is declared as follows:

```
class MY_DECL_SPEC ClassName { ... };
```

Compiling stubs with declaration specifiers

If you are about to package your stubs in a DLL library, compile your C++ stub files, *StubFile*`.cxx`, with a command like the following:

```
cl -DMY_DECLSPEC_EXPORT ... StubFile.cxx
```

By setting the `MY_DECLSPEC_EXPORT` macro on the command line, `_declspec(dllexport)` declarations are inserted in front of the public class declarations in the stub. This ensures that applications will be able to import the public definitions from the stub DLL.

Endpoint References

References provide a convenient and flexible way of identifying and locating specific services.

Introduction to Endpoint References

An endpoint reference is an object that encapsulates addressing information for a particular WSDL service. Essentially, a reference encapsulates all of the information that is required to open a connection to an endpoint. References have the following features:

- A reference encapsulates the data from a `wsdl:service` element.
- References can be sent across the wire as parameters of or as return values from operations.
- References can be passed to client proxy constructors, enabling a client to open a connection to a remote endpoint.
- References are protocol and transport neutral.

Note: From Artix 4.1 onwards, the on-the-wire format of endpoint references has changed, in order to comply with the Web Services Addressing 1.0 - WSDL Binding specification. This might give rise to some interoperability issues, if you require Artix 4.1 programs to interact with older Artix versions. For details, please consult Configuring and Managing Artix Solutions.

Note: In versions of Artix prior to 4.0, references were represented by the proprietary type, `IT_Bus::Reference`. Since version 4.0, however, Artix complies with the WS-Addressing standard for endpoint references. For details of migration issues around references, see “Migration Scenarios” on page 136.

Note: You cannot use references with rpc-encoded bindings, because references contain attributes, which are not compatible with rpc-encoding.

XML representation of a reference

An endpoint reference is represented by the `wsa:EndpointReferenceType` type from the following WS-Addressing schema:

ArtixInstallDir`/schemas/wsaddressing.xsd`

The WS-Addressing schema is also available online at:

http://www.w3.org/2005/08/addressing/ws-addr.xsd

The XML representation is used when marshaling or unmarshaling a reference as a WSDL operation parameter.

C++ representation of a reference

In C++, an endpoint reference is represented by an instance of the `WS_Addressing::EndpointReferenceType` class.

Empty endpoint reference

An endpoint reference containing the following address:

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

represents an *empty endpoint reference* (also called a *null endpoint reference*). You cannot send any messages to such an endpoint reference.

Contents of an endpoint reference

Generally, the on-the-wire XML representation of an endpoint reference has the following form (where `wsa:EndpointReference` is an element of `wsa:EndpointReferenceType` type):

```
<wsa:EndpointReference>
    <wsa:Address>xs:anyURI</wsa:Address>
    <wsa:ReferenceParameters>xs:any*</wsa:ReferenceParameters> ?
    <wsa:Metadata>xs:any*</wsa:Metadata> ?
</wsa:EndpointReference>
```

An endpoint reference encapsulates the following data:

- `wsa:Address`—gives the URI of the endpoint, in whichever format is appropriate for the transport in question. This element must be present.

  > **Note:** Because Artix supports references with multiple endpoints (that is, WSDL ports), the `wsa:Address` element, which supports only one endpoint, is often superseded by the `wsa:Metadata` element, which supports multiple endpoints. If both are present, the `wsa:Metatdata` element takes precedence.

- `wsa:ReferenceParameters`—an optional list of additional parameters that might be needed for establishing a connection to the endpoint (or endpoints).
- `wsa:Metadata`—according to the Web Services Addressing 1.0 - WSDL Binding specification, either or both of the following kinds of metadata can be included in this element:
  - *A reference to WSDL metadata*—this metadata identifies an endpoint whose details are contained either in this `wsa:Metadata` section or in an external WSDL file.
  - *Embedded WSDL metadata*—consists either of a WSDL 2.0 `description` element or a WSDL 1.1 `definitions` element. This element contains a fragment from the WSDL contract describing an endpoint (or endpoints).

The Bank example

Figure 15 shows an overview of the Bank example, illustrating how the Bank service uses references to give a client access to a specific account.

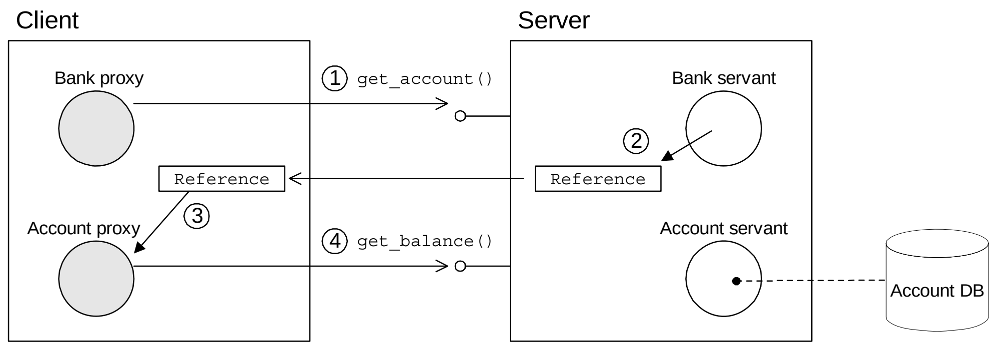

Figure 15: *Using Bank to Obtain a Reference to an Account*

The preceding Bank example can be explained as follows:

1. The client calls `get_account()` on the `BankService` service to obtain a reference to a particular account, *AccName*.
2. The `BankService` creates a reference to the *AccName* account and returns this reference in the response to `get_account()`.
3. The client uses the returned reference to initialize an `AccountClient` proxy.
4. The client invokes operations on the `Account` service through the `AccountClient` proxy.

Using References in WSDL

To use endpoint references in WSDL—that is, to declare operation parameters or return values to be endpoint references—perform the following steps:

1. Define the `wsa` namespace prefix in the `<definitions>` tag at the start of the contract—for example, by setting `xmlns:wsa="http://www.w3.org/2005/08/addressing"`.
2. Import the WS-Addressing schema using an `xsd:import` element.
3. Declare the relevant parameters and return values to be of `wsa:EndpointReferenceType` type.

The WS-Addressing XML schema

The WS-Addressing schema is stored in the following file:

ArtixInstallDir`/schemas/wsaddressing.xsd`

The schema is also available online at:

http://www.w3.org/2005/08/addressing/ws-addr.xsd

WS-Addressing namespace URI

The endpoint reference type is defined in the following target namespace:

`http://www.w3.org/2005/08/addressing`

To access the WS-Addressing types in a WSDL contract file, you should introduce a namespace prefix in the `<definitions>` tag, as follows:

```
<definitions xmlns="..."
    xmlns:wsa="http://www.w3.org/2005/08/addressing"
     ... >
```

Endpoint reference type

The WS-Addressing schema defines an *endpoint reference type* for use within WSDL contracts. The endpoint reference type is, as follows:

WSAPrefix`:EndpointReferenceType`

Where *WSAPrefix* is associated with the `http://www.w3.org/2005/08/addressing` namespace URI:

The Bank WSDL contract

Example 69 shows the WSDL contract for the Bank example that is described in this section. There are two port types in this contract, `Bank` and `Account`. For each of the two port types there is a SOAP binding, `BankBinding` and `AccountBinding`.

Example 69: *Bank WSDL Contract*

```
  <?xml version="1.0" encoding="UTF-8"?>
1 <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>
2         <xsd:import schemaLocation="/schemas/wsaddressing.xsd"
     namespace="http://www.w3.org/2005/08/addressing"/>
          <schema elementFormDefault="qualified"
```

Example 69: *Bank WSDL Contract*

```
     targetNamespace="http://www.iona.com/bus/demos/bank"
            xmlns="http://www.w3.org/2001/XMLSchema">
            <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">
3       <part name="return" type="wsa:EndpointReferenceType"/>
    </message>

    <message name="get_account">
        <part name="account_name" type="xsd:string"/>
    </message>
    <message name="get_accountResponse">
4       <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>

5       <operation name="create_account">
            <input name="create_account"
                   message="tns:create_account"/>
```

Example 69: *Bank WSDL Contract*

```
            <output name="create_accountResponse"
                    message="tns:create_accountResponse"/>
        </operation>

6       <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>

    <binding name="BankBinding" type="tns:Bank">
        ...
    </binding>
    <binding name="AccountBinding" type="tns:Account">
        ...
    </binding>

7   <service name="BankService">
        <port name="BankPort" binding="tns:BankBinding">
            <soap:address
            location="http://localhost:0/BankService/BankPort/"/>
        </port>
    </service>
    <service name="BankServiceRouter">
        <port name="BankPort" binding="tns:BankBinding">
            <soap:address
              location="http://localhost:0/BankService/BankPort/"/>
        </port>
    </service>
8   <service name="AccountService">
        <port name="AccountPort" binding="tns:AccountBinding">
            <soap:address location="http://localhost:0" />
        </port>
    </service>
</definitions>
```

The preceding WSDL contract can be described as follows:

1. The `<definitions>` tag associates the `wsa` prefix with the `http://www.w3.org/2005/08/addressing` namespace URI. This means that the reference type is identified as `wsa:EndpointReferenceType`.
2. The `xsd:import` imports the `wsa:EndpointReferenceType` type definition from the WS-Addressing schema, `wsaddressing.xsd`. You must edit this line if the references schema is stored at a different location relative to the bank WSDL file. Artix stores the WS-Addressing schema at *ArtixInstallDir*`/schemas/wsaddressing.xsd`.

   Note: Alternatively, you could cut and paste the references schema directly into the WSDL contract at this point, replacing the `xsd:import` element.

3. The `create_accountResponse` message (which is the `out` parameter of the `create_account` operation) is defined to be of `wsa:EndpointReferenceType` type.
4. The `get_accountResponse` message (which is the `out` parameter of the `get_account` operation) is defined to be of `wsa:EndpointReferenceType` type.
5. The `create_account` operation defined on the `Bank` port type is defined to return a `wsa:EndpointReferenceType` type.
6. The `get_account` operation defined on the `Bank` port type is defined to return a `wsa:EndpointReferenceType` type.
7. The information contained in this `<service name="BankService">` element is approximately the same as the information that is held in a `BankService` reference, apart from the addressing information in the `soap:address` element.

   The `BankService` reference generated at runtime replaces the `http://localhost:0/BankService/BankPort/` SOAP address with `http://`*host_name*`:`*IP_port*`/BankService/BankPort/` where *host_name* and *IP_port* are substituted with the port address that the server is actually listening on (dynamic port allocation).

   Note: If the IP port in the WSDL contract is non-zero, Artix uses the specified port instead of performing dynamic port allocation. The hostname would still be substituted, however.

8. The information contained in this `<service name="AccountService">` element serves as a prototype for generating `AccountService` references.

   Because the account objects are registered as transient servants, the corresponding `AccountService` references are cloned from the `AccountService` service at runtime by altering the following data:

- The service QName is replaced by a transient service QName, which consists of `AccountService` concatenated with a unique ID code.
- The `http://localhost:0` SOAP address is replaced by `http://`*host_name*`:`*IP_port*`/`*TransientURLSuffix*, where *host_name* and *IP_port* are set to the port address that the server is listening on and *TransientURLSuffix* is a suffix that is unique for each transient reference.

Programming with References

This section explains how to program with endpoint references, using a simple bank application as a source of examples. The bank server supports a `create_account()` operation and a `get_account()` operation, which return references to `Account` objects.

To program with references, you need to know how to generate references on the server side and how to resolve references on the client side.

Creating References

This subsection describes how to create endpoint references, which can be generated on the server side in order to advertise the location of a service to clients.

The following topics are discussed in this section:

- Factory pattern.
- Creating a reference from a static servant.
- Creating a reference from a transient servant.
- Creating a reference from a WSDL contract.
- Creating an empty reference.

Factory pattern

References are usually created in the context of a *factory pattern*. This pattern involves at least two kinds of object:

- The *product*—that is, the type of object to which the references refer.
- The *factory*—which generates references to the first type of object.

For example, the Bank is a factory that generates references to Accounts.

Creating a reference from a static servant

Example 70 shows how to create a `BankService` reference from a static servant, `BankImpl`.

Example 70: *Creating a Reference from a Static Servant*

```
// C++
...
try {
    IT_Bus::Bus_var bus = IT_Bus::init(argc, (char **)argv);

    IT_Bus::QName service_name(
        "", "BankService", "http://www.iona.com/bus/demos/bank"
    );

1   BankImpl my_bank(bus);

2   IT_WSDL::WSDLService* wsdl_service =
        get_bus()->get_service_contract(service_name);
3   bus->register_servant(
        my_bank,
        *wsdl_service
    );

4   IT_Bus::Service_var service =
   bus->get_service(service_name);

5   WS_Addressing::EndpointReferenceType bank_reference;
    service->get_endpoint_reference(bank_reference);
    ...
}
```

The preceding C++ code can be described as follows:

1. This line creates a `BankImpl` servant instance, which implements the `Bank` port type.
2. Call the `IT_Bus::Bus::get_service_contract()` function to find details of the `service_name` service amongst the known WSDL contracts. This function returns a parsed WSDL service element, of `IT_WSDL::WSDLService` type.
3. The `register_servant()` function registers a static servant instance, taking the following arguments:
    - Servant instance.
    - Parsed WSDL service element.

    Note: The preceding example activates all of the ports associated with the Bank service. If you want to activate ports individually, see “Activate ports individually” on page 55.

    The return value is an `IT_Bus::Service` object, which references the original `BankService` WSDL service.
4. Call `IT_Bus::Bus::get_service()` to get a pointer to the `Service` object.

5. The get_endpoint_reference() function populates an endpoint reference, based on the state of the service object, service.

> **Note:** In versions of Artix prior to 4.0, the equivalent functionality (a function that returns an IT_Bus::Reference type) was provided by the get_reference() function.

Creating a reference from a transient servant

Example 71 gives the implementation of the BankImpl::create_account(), function which shows how to create an AccountService reference from a transient servant, AccountImpl.

Example 71: *Creating a Reference from a Transient Servant*

```
// C++
void
BankImpl::create_account(
    const IT_Bus::String &account_name,
    WS_Addressing::EndpointReferenceType &account_reference
) IT_THROW_DECL((IT_Bus::Exception))
{
    AccountMap::iterator account_iter = m_account_map.find(
                                            account_name
                                        );
    if (account_iter == m_account_map.end())
    {
        cout << "Creating new account: "
             << account_name.c_str() << endl;

1       AccountImpl * new_account = new AccountImpl(
            get_bus(), account_name, 0
        );

2       IT_WSDL::WSDLService* wsdl_template_service =
            get_bus()->get_service_contract(
                AccountImpl::SERVICE_NAME
            );

3       IT_Bus::Service_var cloned_service =
            get_bus()->register_transient_servant(
                *new_account,
                *wsdl_template_service
            );

        // Now put the details for the account into the map so
        // we can retrieve it later.
        //
        AccountDetails details;
4       details.m_service = cloned_service.release();
        details.m_account = new_account;

        account_iter = m_account_map.insert(
            AccountMap::value_type(account_name, details)
        ).first;
    }
```

Example 71: *Creating a Reference from a Transient Servant*

```
5       (*account_iter).second.m_service->get_endpoint_reference(
                account_reference
            )
}
```

The preceding C++ code can be described as follows:

1. This line creates an `AccountImpl` servant instance, which implements the `Account` port type.
2. Call the `IT_Bus::Bus::get_service_contract()` function to find details of the `AccountImpl::SERVICE_NAME` service amongst the known WSDL contracts. This function returns a parsed WSDL service element, of `IT_WSDL::WSDLService` type.
3. The `register_transient_servant()` function registers a transient servant instance, taking the following arguments:
   - Servant instance.
   - Parsed WSDL service element.

   > **Note:** The preceding example activates all of the ports associated with the Bank service. If you want to activate ports individually, see "Activate ports individually" on page 61.

   The return value is an `IT_Bus::Service` object, which references a WSDL service cloned from the `AccountService` template service.
4. The `release()` function is part of the Artix smart pointer API—it tells the smart pointer, `cloned_service`, not to delete the referenced `IT_Bus::Service` object once the `cloned_service` smart pointer goes out of scope.
5. The `get_endpoint_reference()` function populates an endpoint reference, based on the state of the account service object.

   > **Note:** In versions of Artix prior to 4.0, the equivalent functionality (a function that returns an `IT_Bus::Reference` type) was provided by the `get_reference()` function.

Creating a reference from a WSDL contract

You can create a reference directly from an IT_WSDL::WSDLService object, which is the Artix representation of a parsed wsdl:service element. Call the IT_Bus::Bus::populate_endpoint_reference() function as follows:

```
// C++
IT_Bus::QName service_qname("", ..., ...);

const WSDLService * wsdl_service =
   bus->get_service_contract(service_qname);

WS_Addressing::EndpointReferenceType   result;

bus->populate_endpoint_reference(
    *wsdl_service,
    result
);
```

As this example shows, you can create an endpoint reference without ever registering a servant.

Creating an empty reference

You can create an empty or null reference as follows:

```
// C++
WS_Addressing::EndpointReferenceType null_reference;
null_reference.getAddress().getvalue().set_uri(
    "http://www.w3.org/2005/08/addressing/none"
);
```

Resolving References

To a client, an WS_Addressing::EndpointReferenceType object is just an opaque token that can be used to open a connection to a particular Artix service. The basic usage pattern on the client side, therefore, is for the client to obtain a reference from somewhere and then use the reference to initialize a proxy object.

Initializing a client proxy with a reference

Client proxies include special constructors to initialize the proxy from an WS_Addressing::EndpointReferenceType object. For example, the AccountClient proxy class includes the following constructors:

```
// C++
AccountClient(
    const WS_Addressing::EndpointReferenceType & epr_ref,
    IT_Bus::Bus_ptr bus = 0
);

AccountClient(
    const WS_Addressing::EndpointReferenceType& epr_ref,
    const IT_Bus::String&                       wsdl_location,
    const IT_Bus::QName&                        service_name,
    const IT_Bus::String&                       port_name,
    IT_Bus::Bus_ptr bus = 0
);
```

The first form of constructor connects to the first port in the reference.

The second form of constructor is useful, if the reference contains multiple ports. You can use the port_name argument to specify which port the client connects to, while leaving the wsdl_location and service_name arguments empty. For example, to initialize a proxy that connects to the CORBAPort port from the multi_port_epr endpoint reference, call the constructor as follows:

```
// C++
AccountClient* proxy = new AccountClient(
    multi_port_epr,
    IT_Bus::String::EMPTY,
    IT_Bus::QName::EMPTY_QNAME,
    "CORBAPort"
);
```

The second form of constructor is also useful for interoperability purposes, where an endpoint reference originates from a non-Artix application. The WS-Addressing specification does *not* require an endpoint reference to encapsulate metadata for the endpoint. Hence, sometimes the endpoint reference might contain just an URL (the endpoint address) and provide no other details about the endpoint. In this case, you can supply the missing endpoint details directly from a WSDL contract. The second form of constructor enables you to specify the WSDL contract location, wsdl_location, the service QName, service_name, and port name, port_name, for the endpoint.

Client example

Example 72 shows some sample code from a client that obtains a reference to an `Account` and then uses this reference to initialize an `AccountClient` proxy object.

Example 72: *Client Using an Account Reference*

```
// C++
...
BankClient bankclient;

// 1. Retrieve an account reference from the remote Bank object.
WS_Addressing::EndpointReferenceType account_reference;
bankclient.get_account("A. N. Other", account_reference);

// 2. Resolve the account reference.
AccountClient account(account_reference);

IT_Bus::Float balance;
account.get_balance(balance);
```

The WSDL Publish Plug-In

It is recommended that you activate the *WSDL publish plug-in* for any applications that generate and export references. To use references, the client must have access to the WSDL contract referred to by the reference. The simplest way to accomplish this is to use the `wsdl_publish` plug-in.

By default, a reference's WSDL location URL would reference a local file on the server system. This suffers from the following drawbacks:

- Clients are not able to access the server's WSDL file, unless they happen to share the same file system.
- Endpoint information (the physical contract) might be inaccurate or incomplete, because the server updates transport properties at runtime.

In both of these cases, the client needs to have a way of obtaining the dynamically-updated WSDL contract directly from the remote server. The simplest way to achieve this is to configure the server to load the WSDL publish plug-in. The WSDL publish plug-in automatically opens a HTTP port, from which clients can download a copy of the server's in-memory WSDL model.

Loading the WSDL publish plug-in

To load the WSDL publish plug-in, edit the `artix.cfg` configuration file and add `wsdl_publish` to the `orb_plugins` list in your application's configuration scope. For example, if your application's configuration scope is `demos.server`, you might use the following `orb_plugins` list:

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

Generating references without the WSDL publish plug-in

Figure 16 gives an overview of how a reference is generated when the WSDL publish plug-in is *not* loaded.

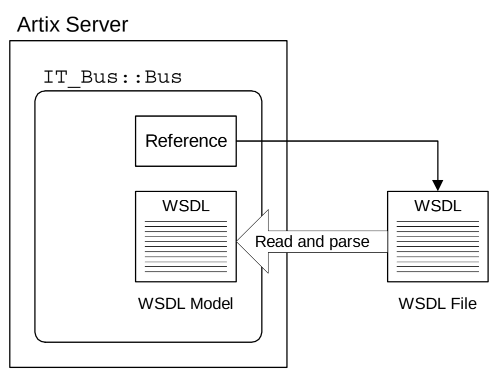

Figure 16: *Generating References without the WSDL Publish Plug-In*

In this case, references generated by the `IT_Bus::Bus` object would, by default, have their WSDL location set to point at the local WSDL file.

The Artix server reads and parses the WSDL file as it starts up, creating a WSDL model in memory. Because the WSDL model can be updated dynamically by the server, there may be some significant differences between the WSDL model and the WSDL file.

WSDL model

When an Artix server starts up, it reads the WSDL files needed by the registered services—for example, in Figure 16, a single WSDL file is read and parsed. After parsing, the WSDL definitions exist in memory in the form of a *WSDL model*. The WSDL model is an XML parse tree containing all the WSDL definitions imported into a particular `IT_Bus::Bus` instance at runtime. Different `IT_Bus::Bus` instances have distinct WSDL models.

The WSDL model is dynamically updated by the Artix server to reflect changes in the physical contract at runtime. For example, if the server dynamically allocates an IP port for a particular port on a WSDL service, the port's addressing information is updated in the WSDL model.

Generating references with the WSDL publish plug-in

When the WSDL publish plug-in is loaded, the Artix server opens a HTTP port which it uses to publish the in-memory WSDL model. Figure 17 gives an overview of how an Artix reference is generated when the WSDL publish plug-in is loaded.

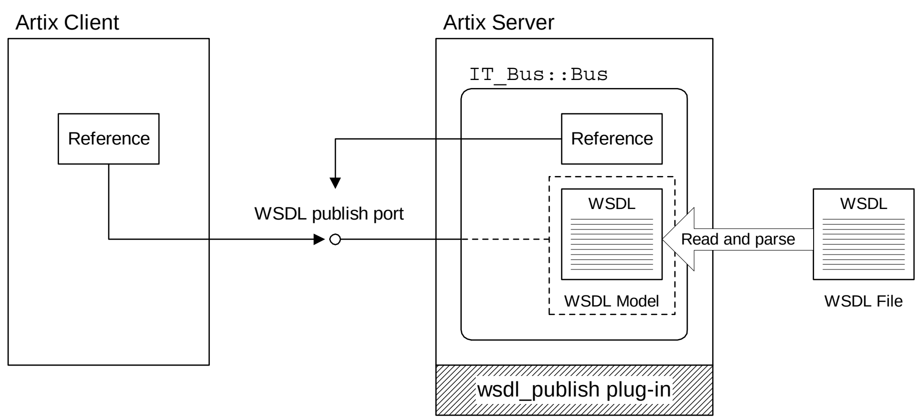

Figure 17: *Generating References with the WSDL Publish Plug-In*

In this case, references generated by the `IT_Bus::Bus` object have their WSDL location set to the following URL:

`http://`*host_name*`:`*WSDL_publish_port*`/`*QueryString*

Where *host_name* is the server host, *WSDL_publish_port* is an IP port used specifically for the purpose of serving up WSDL contracts, and *QueryString* is a string that requests a particular WSDL contract (see "Querying the WSDL publish port" on page 135).

If a client accesses the WSDL location URL, the server will convert the WSDL model to XML on the fly and return the resulting WSDL contract in a HTTP message.

Specifying the WSDL publish port

If you need to specify the WSDL publish port explicitly, set the `plugins:wsdl_publish:publish_port` variable in the Artix configuration file.

Querying the WSDL publish port

It is possible to query the WSDL publish port to obtain various kinds of metadata for the services currently running in the server. Details of this query protocol are provided in *Configuring and Deploying Artix Solutions*.

Usefulness of the published WSDL model

In most cases, clients do not need to download the published WSDL model at all. Published WSDL is primarily useful for *dynamic clients* that try to invoke an operation on the fly. Because dynamic clients are *not* compiled with Artix stub code, the only way they can obtain the logical contract is by downloading the published WSDL model.

Whether or not you can use the physical part of the WSDL model depends on how the corresponding servant is registered on the server side:

- If registered as static, the physical contract is available from the WSDL model.
- If registered as transient, the physical contract is available only from the reference, not from the WSDL model. The associated reference encapsulates a *cloned service* which is generated at runtime and is not included in the WSDL model. See “Registering Transient Servants” on page 56.

Multiple Bus instances

Occasionally, you might need to create an Artix server with more than one IT_Bus::Bus instance. In this case, you should be aware that separate WSDL models are created for each Bus instance and separate HTTP ports are also opened to publish the WSDL models—see Figure 18.

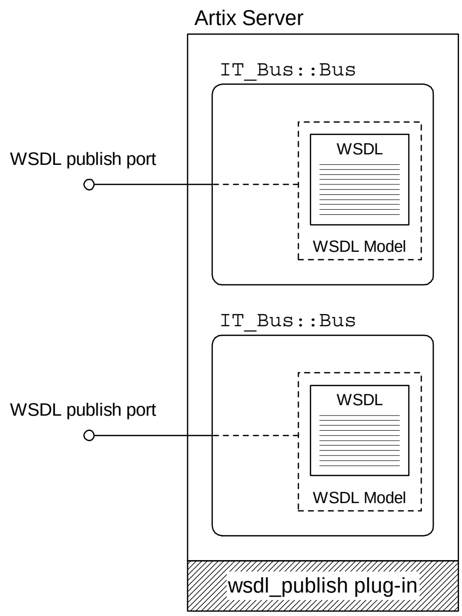

Figure 18: *WSDL Publish Plug-In and Multiple Bus Instances*

Migration Scenarios

With the release of Artix 4.0, Artix switched from using a proprietary reference format to using the standard WS-Addressing endpoint reference format. If you have existing applications that use the old proprietary reference format, you might want to consider migrating those applications to the WS-Addressing standard.

The following migration scenarios are considered here:

- Retaining proprietary references.
- Migrating to WS-Addressing references.
- Mixing new and old references.

Retaining proprietary references

The simplest option for existing applications that are being migrated to Artix 4.0 is to continue using the old Artix proprietary references. Artix 4.0 maintains complete backwards compatibility with the IT_Bus::Reference type. Specifically, the backwards compatibility enables you to leave the following aspects of your application untouched:

- *WSDL contracts*—continue to use the references:Reference type, where the references namespace prefix is associated with the http://schemas.iona.com/references namespace URI.
- *C++ source code*—continue to use the IT_Bus::Reference type.
- *On-the-wire format*—remains the same as Artix 3.0.

Migrating to WS-Addressing references

If you have an existing application that you want to migrate to Artix 4.0, you can switch to the WS-Addressing standard by changing the following aspects of your application:

- *WSDL contracts*—replace the references:Reference type by the wsa:EndpointReferenceType type, where the wsa namespace prefix is associated with the http://www.w3.org/2005/08/addressing namespace URI.

  Modify the xsd:import element for references so that it imports the new WS-Addressing schema instead of the old Artix references schema. For example:

```
<definitions xmlns="..."
    xmlns:wsa="http://www.w3.org/2005/08/addressing"
    ... >
    <types>
      <xsd:import schemaLocation="/schemas/wsaddressing.xsd"
        namespace="http://www.w3.org/2005/08/addressing"/>
       ...
    </types>
    ...
</definitions>
```

- *C++ source code*—besides regenerating Artix stub code from the updated WSDL contracts, two changes are required:
  - Replace the IT_Bus::Reference type by the WS_Addressing::EndpointReferenceType type.
  - Replace any occurrence of IT_Bus::Service::get_reference() with IT_Bus::Service::get_endpoint_reference(), where get_endpoint_reference() populates an endpoint reference argument instead of returning an endpoint reference.
- *On-the-wire format*—the endpoint reference is formatted as a wsa:EndpointReference element (which is of wsa:EndpointReferenceType type).

Mixing new and old references

It is possible to mix the new and old reference types in a single program.

- *Using new and old references in the same program*—you can mix new and old style references freely in the same program. Parameters declared to be of wsa:EndpointReferenceType type in WSDL will map to the `WS_Addressing::EndpointReferenceType` C++ type and parameters declared to be of `references:Reference` type in WSDL will map to the `IT_Bus::Reference` C++ type.

Callbacks

An Artix callback is a pattern, where the client implements a WSDL service. This chapter explains the basic concept of a callback and describes how to implement a simple example.

Overview of Artix Callbacks

What is a callback?

A *callback* is a pattern, where a client implements a service whose operations can be called by a server (the server *calls back* on the client). In other words, the usual direction of the operation invocation is reversed in this case.

Stock monitor scenario

Figure 19 shows an example of a scenario where the callback pattern is used. On the client side, a GUI application is running that is used to monitor and trade stocks and shares. One of the services accessible to the clients is a *Stock Monitor Service* that tracks the price of stocks in real time.

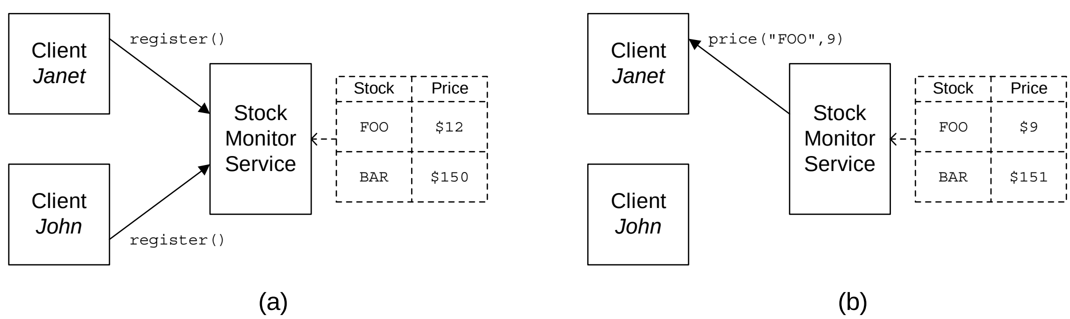

Figure 19: *Callback Pattern Illustrated by a Stock Monitor Scenario*

Scenario description

The stock monitor scenario shown in Figure 19 can be described as follows:

- Two stockbrokers, Janet and John, want to monitor the current price of two stocks, `FOO` and `BAR`. Janet has orders to sell `FOO`, if it dips below $10, and John has orders to sell `BAR`, if it dips below $100.
- When Janet and John log on in the morning, they use the stockbroking application on their PCs to set up price triggers for the respective stocks. As shown in Figure 19 (a), the client application sets up the price trigger by calling the remote `register()` operation on the Stock Monitor Service.

- Later that afternoon, when the stock price of FOO drops to $9, the Stock Monitor Service sends a callback notification to Janet's client application, alerting her to the fact that the price has just dropped below $10—see Figure 19 (b).

Characteristics of the callback pattern

Callback scenarios typically have the following characteristics:

- *Clients must implement a callback service*—the callback service is required, so that clients can receive notifications from the server side. One consequence of this is that implementing a callback client is rather like implementing a server.
- *IP port for callback service is dynamically allocated*—typically, on a client host, it is not possible to allocate a fixed IP port. In most cases, therefore, it is necessary to use a dynamically allocated IP port for the callback service.
- *Clients must register interest in receiving callbacks*—the server must be notified explicitly that the client is available and interested in receiving certain events. In particular, the server needs to acquire the address of the client's callback service.
- *Callbacks typically occur asynchronously*—usually, the server is constantly monitoring some state and must be ready at any time to send a notification to the registered clients. This normally requires the server to be multi-threaded.

  Likewise, the client must be ready to receive a callback at any time from the server. This normally requires the client to be multi-threaded.

Callback demonstration

The callback example described in this section is based on the Artix callback demonstration, which is located in the following directory:

ArtixInstallDir/samples/callbacks/basic_callback

Demonstration scenario

Callbacks rely, essentially, on endpoint references. Using references, the client can encapsulate the details of its callback service and pass on these details to the server in a reference parameter. Figure 20 illustrates how this process works.

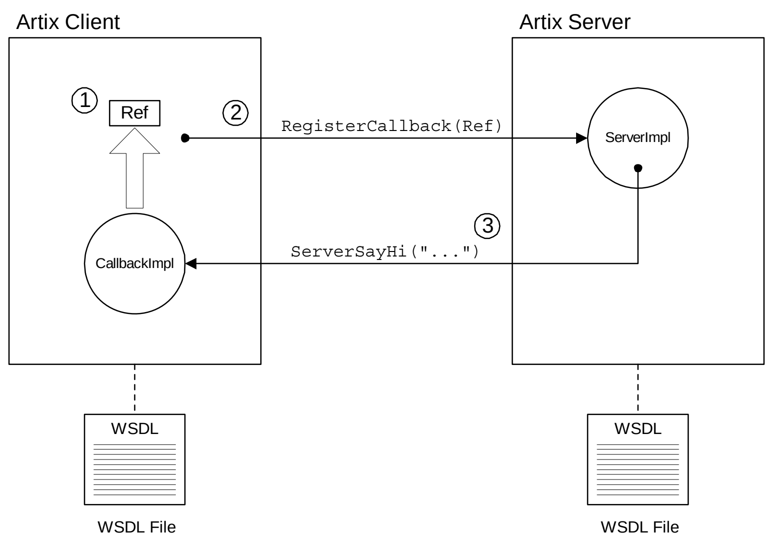

Figure 20: *Overview of the Callback Demonstration*

Callback steps

Figure 20 on page 141 shows the callback proceeding according to the following steps:

1. After the basic initialization steps, including registration of the `CallbackImpl` servant and `CallbackService` service, the client generates a reference for the callback service.

   The client callback service is activated and capable of receiving incoming invocations as soon as it is registered.
2. The client calls `RegisterCallback()` on the remote server, passing the reference generated in the previous step.
3. When the server receives the callback reference, it immediately calls back on the `CallbackImpl` servant by invoking `ServerSayHi()`.

> **Note:** In a more realistic application, it is likely that the server would cache a copy of the callback reference and call back on the client at a later time, instead of calling back immediately.

Callback WSDL Contract

This subsection describes the WSDL contract that defines the interaction between the client and the server in the callback demonstration. This WSDL contract is somewhat unusual in that it defines port types both for the client and for the server applications.

WSDL contract

Example 73 shows the WSDL contract used for the callback demonstration.

Example 73: *Example Callback WSDL Contract*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="basic_callback"
   targetNamespace="http://www.microfocus/com/callback"
         xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:corba="http://schemas.iona.com/bindings/corba"
xmlns:ns1="http://www.microfocus/com/callback/corba/typemap/"
         xmlns:ns2="http://schemas.iona.com/routing"
         xmlns:addressing="http://www.w3.org/2005/08/addressing"
         xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
         xmlns:tns="http://www.microfocus/com/callback"
         xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <types>
    <schema targetNamespace="http://www.microfocus/com/callback"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
           <import
         namespace="http://www.w3.org/2005/08/addressing"
         schemaLocation="../../../../schemas/wsaddressing.xsd"/>

            <element name="callback_message" type="xsd:string"/>
            <element name="RegisterCallback">
                <complexType>
                    <sequence>
                        <element name="reference"
                        type="addressing:EndpointReferenceType"/>
                    </sequence>
                </complexType>
            </element>
            <element name="returnType" type="xsd:string"/>
        </schema>
    </types>

    <message name="server_sayHi">
        <part element="tns:callback_message"
              name="return_message"/>
    </message>
    <message name="register_callback">
        <part element="tns:RegisterCallback"
              name="callback_object"/>
    </message>
    <message name="returnMessage">
        <part element="tns:returnType" name="the_return"/>
    </message>

1   <portType name="CallbackPortType">
        <operation name="ServerSayHi">
            <input message="tns:server_sayHi"
                   name="ServerSayHiRequest"/>
            <output message="tns:returnMessage"
                    name="ServerSayHiResponse"/>
        </operation>
```

Example 73: *Example Callback WSDL Contract*

```
     </portType>

2    <portType name="ServerPortType">
         <operation name="RegisterCallback">
             <input message="tns:register_callback"
                    name="RegisterCallbackRequest"/>
             <output message="tns:returnMessage"
                     name="RegisterCallbackResponse"/>
         </operation>
     </portType>
     ...
     <service name="CallbackService">
         <port binding="tns:CallbackPortType_SOAPBinding"
               name="CallbackPort">
3            <soap:address location="http://localhost:0"/>
         </port>
     </service>

     <service name="SOAPService">
         <port binding="tns:ServerPortType_SOAPBinding"
               name="SOAPPort">
4            <soap:address location="http://localhost:9000"/>
         </port>
     </service>
 </definitions>
```

The preceding WSDL contract can be described as follows:

1. The `CallbackPortType` port type is implemented on the client side and supports a single WSDL operation:
   - `ServerSayHi` operation—takes a single string argument. The server calls back on this operation after it has received a reference to the client's service.
2. The `ServerPortType` port type is implemented on the server side and supports a single WSDL operation:
   - `RegisterCallback` operation—takes a single endpoint reference argument, which is used to pass a reference to the client callback object.
3. The client callback address should be specified as `http://localhost:0`, which acts as a placeholder for the address generated dynamically at runtime. When the callback servant is activated, Artix modifies the address, replacing `localhost` by the client's hostname and replacing `0` by a randomly allocated IP port number.

   > **Note:** Do *not* add a terminating `/` character at the end of the address—for example, `http://localhost:0/`. Artix does not accept addresses terminated with a forward slash.

4. The server's address, `http://`*SvrHost*`:`*SvrPort*, should be specified explicitly, where *SvrHost* is the host where the server is running and *SvrPort* is a fixed IP port. In this example, the client obtains the server's address directly from the WSDL contract file.

Client Implementation

In a callback scenario, the client plays a hybrid role: part client, part server. Hence, the implementation of the callback client includes coding steps you would normally associate with a server, including an implementation of a servant class. The callback client implementation consists of two main parts, as follows:

- Client main function.
- CallbackImpl servant class.

Client main function

Example 74 shows the code for the callback client main function, which instantiates and registers a `CallbackImpl` servant before calling on the remote server to register the callback.

Example 74: *Callback Client Main Function*

```
#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_cal/iostream.h>

#include "ServerClient.h"
#include "CallbackImpl.h"

IT_USING_NAMESPACE_STD

using namespace BasicCallback;
using namespace IT_Bus;
using namespace WS_Addressing;

int
main(int   argc, char* argv[])
{
    try
    {
        // Need to retain reference to Bus
        //
        Bus_var bus = IT_Bus::init(argc, argv);

        QName soap_service_qname(
            "",
            "SOAPService",
            "http://www.microfocus/com/callback"
        );

        ServerClient client(
            "../../etc/basic_callback.wsdl",
            soap_service_qname,
            "SOAPPort",
            bus
        );

1       CallbackImpl servant(bus);
```

Example 74: *Callback Client Main Function*

```
2           QName service_qname(
                "",
                "CallbackService",
                "http://www.microfocus/com/callback"
            );

            // Use Bus reference to register and activate servant
            //

3           Service_var service =
                bus->register_transient_servant(
                    servant,
                    "../../etc/basic_callback.wsdl",
                    service_qname
                );

            EndpointReferenceType callback_reference;
4           service->get_endpoint_reference(callback_reference);

            String outcome;

            // Create instance of wrapper class
            //
            RegisterCallback callback_object;

            // Set reference into wrapper
            //
            callback_object.setreference(callback_reference);

5           client.RegisterCallback(callback_object, outcome);

            // Display return message from RegisterCallback operation
            //
            cout << "\t" << outcome << endl;

            bus->shutdown(true);
        }
        catch (const IT_Bus::Exception& e)
        {
            cout << endl << "Error : Unexpected error occured!"
                 << endl << e.message()
                 << endl;
            return -1;
        }
        return 0;
    }
```

The preceding code example can be explained as follows:

1. The CallbackImpl servant class implements the CallbackPortType port type. The CallbackImpl instance created on this line is the client callback object.
2. The service_qname specifies the WSDL service to be activated on the client side. This QName refers to the <service name="CallbackService"> element in Example 73 on page 142.

3. Register the callback servant with the Bus, thereby activating the `CallbackService` service. From this point on, the `CallbackService` service is active and able to process incoming callback requests in a background thread.
4. A reference to the callback service is generated by calling `IT_Bus::Service::get_endpoint_reference()`.
5. This line invokes the `RegisterCallback()` operation on the remote server, passing in the reference to the client callback object. Before this operation returns, the server calls back on the `ServerSayHi()` operation of the `CallbackImpl` servant.

CallbackImpl servant class

Example 75 shows the implementation of the `CallbackImpl` servant class, which is responsible for receiving the `CallbackImpl::ServerSayHi()` callback from the server. The implementation of this servant class is trivial. It follows the usual pattern for a servant class implementation and the `ServerSayHi()` function simply prints out its string argument.

Example 75: *CallbackImpl Servant Class Implementation*

```
#include "CallbackImpl.h"
#include <it_cal/cal.h>

IT_USING_NAMESPACE_STD
using namespace BasicCallback;

CallbackImpl::CallbackImpl(IT_Bus::Bus_ptr bus) :
   CallbackServer(bus)
{
}

CallbackImpl::~CallbackImpl()
{
}

IT_Bus::Servant*
CallbackImpl::clone() const
{
    return new CallbackImpl(get_bus());
}

void
CallbackImpl::ServerSayHi(
    const IT_Bus::String &return_message,
    IT_Bus::String &the_return
) IT_THROW_DECL((IT_Bus::Exception))
{
    // User code goes in here
    cout <<"\t\tCallbackImpl::ServerSayHi() called"<<endl;
    cout << "\t\t" << return_message <<endl;
    cout <<"\t\tCallbackImpl::ServerSayHi() ended"<<endl;
    the_return = "The callback was successful";
}
```

Server Implementation

The implementation of the server in this callback example follows the usual pattern for an Artix server. The server main function instantiates and registers a servant object. A separate file contains the implementation of the servant class, `ServerImpl`. The server implementation thus consists of two main parts, as follows:

- Server main function.
- ServerPortType implementation.

Server main function

Example 76 shows the code for the server main function, which instantiates and registers a `ServerImpl` servant. The server then waits for the client to register a callback using the `RegisterCallback` operation.

Example 76: *Server Main Function*

```
#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_bus/fault_exception.h>
#include <it_cal/iostream.h>

IT_USING_NAMESPACE_STD

#include "ServerImpl.h"

using namespace BasicCallback;
using namespace IT_Bus;

int
main(int argc, char* argv[])
{
    try
    {
       IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);

1      ServerImpl servant(bus);

2      IT_Bus::QName service_name(
        "", "SOAPService", "http://www.microfocus/com/callback"
       );

3       bus->register_servant(
                             servant,
                             "../../etc/basic_callback.wsdl",
                             service_name
                             );

        cout << "Server Ready" << endl;

4       bus->run();
    }
    catch(IT_Bus::Exception& e)
    {
        cout << "Error occurred: " << e.message() << endl;
```

Example 76: *Server Main Function*

```
        return -1;
    }
    return 0;
}
```

The preceding code example can be explained as follows:

1. The `ServerImpl` servant class implements the `ServerPortType` port type, which supports the `RegisterCallback` operation.
2. The `service_qname` refers to the `<service name="SOAPService">` element in Example 73 on page 142.
3. Register the `ServerImpl` servant with the Bus, thereby activating the `SOAPService` service.
4. Call the blocking `IT_Bus::Bus::run()` function to allow the server application to process incoming requests.

ServerPortType implementation

Example 77 shows the implementation of the `ServerImpl` servant class. There is just one WSDL operation, `RegisterCallback()`, to implement in this class.

Example 77: *ServerImpl Servant Class Implementation*

```
#include "ServerImpl.h"
#include <it_cal/cal.h>
#include "CallbackClient.h"

using namespace WS_Addressing;
using namespace BasicCallback;
IT_USING_NAMESPACE_STD


ServerImpl::ServerImpl(IT_Bus::Bus_ptr bus) : ServerServer(bus)
{
}

ServerImpl::~ServerImpl()
{
}

IT_Bus::Servant*
ServerImpl::clone() const
{
    return new ServerImpl(get_bus());
}

void
ServerImpl::RegisterCallback(
1    const BasicCallback::RegisterCallback &callback_object,
    IT_Bus::String &the_return
) IT_THROW_DECL((IT_Bus::Exception))
{
    try
    {
        // Extract reference from wrapper
```

Example 77: *ServerImpl Servant Class Implementation*

```
        EndpointReferenceType callback_epr =
            callback_object.getreference();

      // Instantiate proxy with reference
2     CallbackClient cc(callback_epr);
      IT_Bus::String a_return;
3      cc.ServerSayHi("Server says Hi to the Client", a_return);
       cout << "\t\t" << a_return << endl;
    }
    catch (IT_Bus::Exception& e)
    {
        cout << "Caught Unexpected Exception "
             << e.message() << endl;
    }
    catch (...)
    {
        cout << "Unknown exception" << endl;
    }
    cout << "\tFinished invoking on Callback Object" << endl;
    cout << "\tServerImpl::RegisterCallback Returning" << endl;
    the_return = "The server processing was successful";
}
```

The preceding code example can be explained as follows:

1. The `RegisterCallback()` function takes an endpoint reference argument, which should be a reference to a callback object.
2. This line creates a client proxy, `cc`, for the `CallbackPortType` port type and initializes it with the callback reference, `callback_object`. The reference, `callback_object`, encapsulates details of the `CallbackService` service.
3. This line invokes the `ServerSayHi()` callback on the client.

   This example, where the callback is invoked within the body of `RegisterCallback()`, is a little bit artificial. In a more typical use case, the server would cache an instance of the callback client proxy and then call back later, in response to some event that is of interest to the client.

Routing and Callbacks

Callbacks are fully compatible with Artix routers. References that pass through a router are automatically *proxified*, if necessary. Proxification means that the router automatically creates a new route for the references that pass through it.

Note: Proxification is not necessary, if the transport protocols along the route are the same. For same protocol routing, proxification is disabled by default.

For example, consider the callback routing scenario shown in Figure 21. In this scenario, a SOAP/HTTP Artix server replaces a legacy CORBA server. As part of a migration strategy, legacy

CORBA clients can continue to communicate with the new server by interposing an Artix router to translate between the IIOP and SOAP/HTTP protocols.

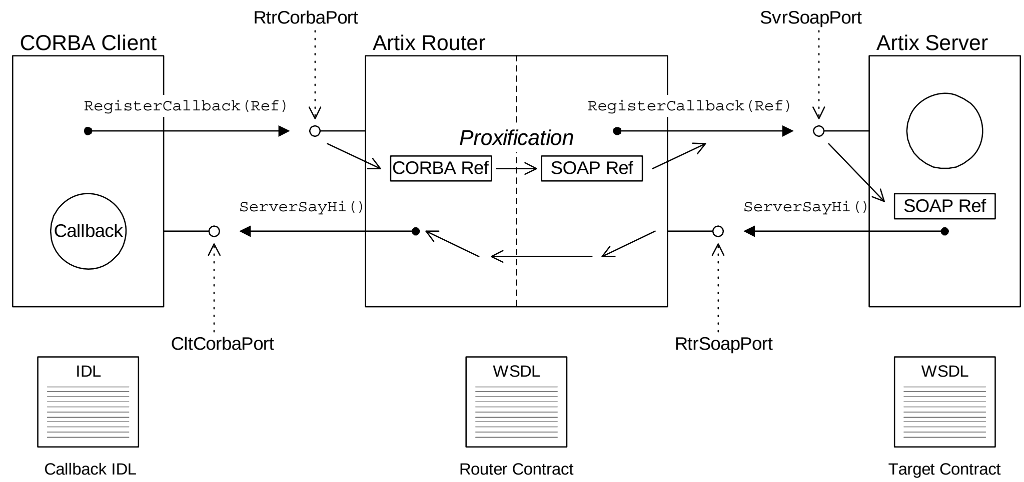

Figure 21: *Overview of a Callback Routing Scenario*

Contracts

The scenario depicted in Figure 21 requires three distinct, but related, contracts as follows:

- Callback IDL.
- Target contract.
- Router contract.

Callback IDL

The CORBA client uses a contract coded in OMG Interface Definition Language (IDL). This IDL contract defines both the target interface (implemented by the Artix server) and the callback interface (implemented by the CORBA client).

Target contract

In this scenario, the target contract is generated from the callback IDL using the IDL-to-WSDL compiler. Hence, this WSDL contract contains both the target interface and the callback interface as WSDL port types.

The target contract also contains a single WSDL service description, which includes the `SvrSoapPort` port.

Router contract

The router contract holds details about the CORBA side of the application as well as the SOAP/HTTP side, including the following information:

- Target WSDL port type.
- Callback WSDL port type.
- CORBA WSDL binding for the target.
- SOAP/HTTP WSDL binding for the target.
- CORBA WSDL service, containing the `RtrCorbaPort` port.
- SOAP/HTTP WSDL service, containing the `SvrSoapPort` port.
- Template SOAP/HTTP WSDL service, needed for generating the transient endpoint with `RtrSoapPort` port.
- Route information.

To specify the location of the generated router contract, you can set the `plugins:routing:wsdl_url` configuration variable in the router scope of the `artix.cfg` configuration file.

Routes

As shown in Figure 21 on page 150, the following routes are created in this scenario:

- *Client-Router-Target route*—this route is documented explicitly in the router contract. The source port, `RtrCorbaPort`, and the destination port, `SvrSoapPort`, are described in the router contract.

  For example, when the client calls the `RegisterCallback()` operation, the request travels initially to the `RtrCorbaPort` on the router (over IIOP) and then on to the `SvrSoapPort` on the target server (over SOAP/HTTP).
- *Target-Router-Client route (callback route)*—the reverse route (for callbacks) is *not* documented explicitly in the router contract. This route is constructed at runtime to facilitate routing callback invocations.

  For example, when the Artix server calls the `ServerSayHi()` callback operation, the request travels to the `RtrSoapPort` on the router (over SOAP/HTTP) and then on to the `CltCorbaPort` on the client (over IIOP).

Proxification

Proxification refers to the process whereby a reference of a certain type (for example, a CORBA reference) that passes through the router is automatically converted to a reference of another type (for example, a SOAP reference).

The proxification process is of key importance to Artix callbacks. If the router in Figure 21 on page 150 did not proxify `RegisterCallback()`'s reference argument, it would be impossible for the server to call back on the client. The server can call back *only* on SOAP/HTTP endpoints, not on IIOP endpoints.

In Figure 21 on page 150, the router proxifies the callback reference as follows:

1. When the `RegisterCallback()` operation is invoked, the router recognizes that the reference argument must be converted into a SOAP/HTTP-format reference.
2. The router dynamically creates a new service and port, `RtrSoapPort`, to receive callback requests in SOAP/HTTP format. The new service is a transient service cloned from a service in the router WSDL contract. The router looks for a template service that satisfies the following criteria:
   - Supports the same port type as the original reference.
   - Supports the same type of binding (for example, SOAP or CORBA) as the target server.

   > **Note:** Artix selects the first service in the WSDL contract that satisfies these criteria. Hence, if more than one service matches the criteria, you must ensure that the template service precedes the other services in the contract file.

3. The router creates a new SOAP/HTTP reference, encapsulating details of the `RtrSoapPort` endpoint.
4. The router forwards the `RegisterCallback()` operation on to the target server in SOAP format, with the proxified SOAP/HTTP reference as its argument.
5. The router dynamically constructs a callback route, with source port, `RtrSoapPort`, and destination port, `CltCorbaPort`.

Enabling proxification for same protocol routing

The router can be used to redirect messages of the same protocol type (for example, SOAP to SOAP). In this case, you can either enable or disable proxification by setting the following variable in the router configuration:

```
plugins:router:use_pass_through = "Boolean";
```

If *Boolean* is `true` (the default), proxification is disabled for same-protocol routing; if `false`, proxification is enabled for same-protocol routing.

When the router is used as a bridge between different protocols (for example CORBA to SOAP), proxification is *always* enabled. It is not possible to disable proxification in this case.

Artix Contexts

Artix contexts are used for the following purposes: to configure Artix transports, bindings and interceptors; and to send extra data in request headers or reply headers.

Introduction to Contexts

This section provides a conceptual overview of Artix contexts, including a brief look at the programming interface required for using contexts with different binding types.

Request, Reply and Configuration Contexts

Artix contexts provide a general purpose mechanism for configuring Artix plug-ins. Contexts enable you to configure both the client-side settings and the server-side settings.

Currently, contexts are used mainly to program transport settings (overriding the settings that appear in the corresponding WSDL port element). Figure 22 gives an overview of the context architecture, where the contexts can be used to modify the attributes of a transport plug-in.

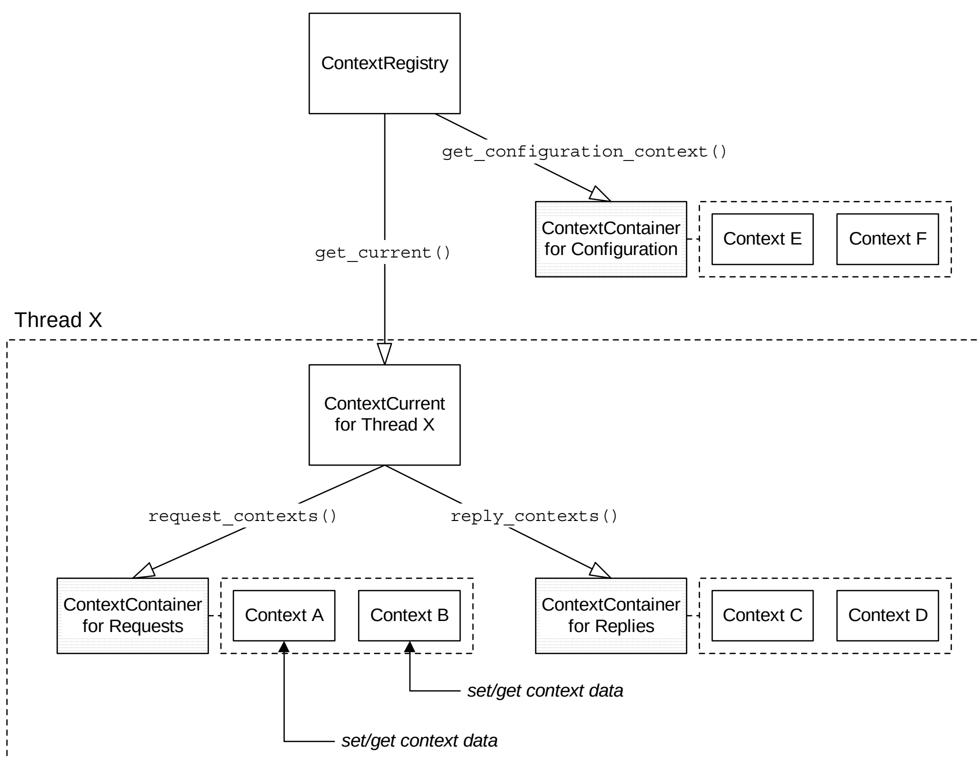

Figure 22: *Overview of the Context Architecture*

Thread affinity

The threading properties of a context depend on the kind of context, as follows:

- *Request and reply contexts*—are held in thread-specific storage, so that different threads can be programmed with different attributes. The root object for obtaining thread-specific data is the `IT_Bus::ContextCurrent` object.
- *Configuration contexts*—are *not* thread-specific.

Request contexts

Request contexts are used to read or modify attributes as follows:

- *On the client side*—setting transport attributes and setting header contexts for outgoing requests.
- *On the server side*—reading header contexts from incoming requests.

By calling the `IT_Bus::ContextCurrent::request_contexts()` function, you can obtain a copy of an `IT_Bus::ContextContainer` object, which contains references to all of the current request contexts.

Reply contexts

Reply contexts are used to read or modify attributes as follows:

- *On the client side*—reading header contexts from incoming replies.
- *On the server side*—setting transport attributes and setting header contexts for outgoing replies.

By calling the `IT_Bus::ContextCurrent::reply_contexts()` function, you can obtain a copy of an `IT_Bus::ContextContainer` object, which contains references to all of the current reply contexts.

Configuration contexts

Configuration contexts are used to read and modify endpoint-specific context data that can be set *before* a connection has initialized. Currently, Artix supports just the following configuration context properties:

- HTTP endpoint URL,
- JMS broker connection security information,
- FTP connection settings.

By calling the `IT_Bus::ContextRegistry::get_configuration_context()` function, you can obtain a copy of an `IT_Bus::ContextContainer` object, which contains references to all of the configuration contexts.

Schema-based API

The API for getting and setting the attributes of a particular context type is generated from an XML schema. The code for a context type is generated by the Artix WSDL-to-C++ compiler as part of the stub code. There are two ways of getting hold of the context stub code, depending on whether the context is a custom type or a built-in type, as follows:

- *Custom context*—for a context that you define yourself you can generate the context stub code by running the WSDL-to-C++ compiler on the context schema file, `CustomContext.xsd`. The stub code then consists of the files `CustomContext_xsdTypes.h`, `CustomContext_xsdTypes.cxx`, `CustomContext_xsdTypesFactory.h` and `CustomContext_xsdTypesFactory.cxx`.
- *Built-in context*—for an Artix-defined context, the stub code is packaged in the Artix library, `it_context_attribute[.lib][.so][.sl]`.

Header Contexts

Artix *header contexts* provide a general purpose mechanism for embedding data in message headers. Currently, you can embed context data in the following types of protocol header:

- SOAP.
- CORBA.

SOAP

When you register a context as a SOAP context (using the appropriate form of the `ContextRegistry::register_context()` function), the corresponding context data is embedded in a SOAP header, as shown in Figure 23.

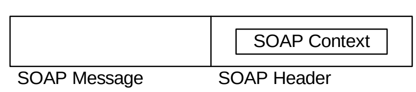

Figure 23: *Inserting Context Data into a SOAP Header*

The context data is sent in an Artix-specific SOAP header.

CORBA

When you register a context as a CORBA context (using the appropriate form of the `ContextRegistry::register_context()` function), the corresponding context data is embedded within a CORBA header as a GIOP service context—see Figure 24.

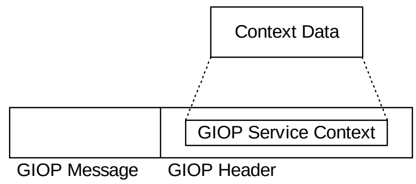

Figure 24: *Inserting Context Data into a GIOP Service Context*

In CORBA, the message formats are defined by the General Inter-ORB Protocol (GIOP) specification. In particular, the GIOP request and reply message formats allow you to include arbitrary header data in GIOP service contexts. Artix creates one GIOP service context for each Artix context. The type of GIOP service context is identified by an IOP context ID, which you specify when registering the Artix context.

Registering Contexts

You register a context type by calling a `register_context()` function on a context registry instance, passing the context name and context type as arguments. The main effect of registering a context type is that the context container adds a type factory reference to an internal table. This type factory reference enables the context container to create context data instances whenever they are needed.

Note: This pre-supposes that the application is linked with the context schema stub code, which creates static instances of the relevant type factories. See "Schema-based API" on page 155.

Getting a context registry instance

To get a reference to a context registry instance, you call the `IT_Bus::Bus::get_context_registry()` function, as in Example 78.

Example 78: *The IT_Bus::Bus::get_context_registry() Function*

```
// C++
namespace IT_Bus {
    class IT_BUS_API Bus
    {
      public:
        virtual ContextRegistry*
        get_context_registry() = 0;
        ...
    };
};
```

Registering a context

In practice, you would seldom need to register a context unless you are implementing your own Artix plug-in. All of the standard Artix contexts are pre-registered (see "Getting and Setting Transport Attributes" on page 206).

You can register request, reply, and configuration contexts in either of the following ways:

- Registering a serializable context.
- Registering a non-serializable context.

Registering a serializable context

A serializable context is a data type that inherits from the `IT_Bus::AnyType` base class. Example 79 shows the signature of the `register_context()` function in the `IT_Bus::ContextRegistry` class, which is used to register a serializable context.

Example 79: *The register_context() Function for Serializable Contexts*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API ContextRegistry
    {
      public:
        enum ContextType {
            TYPE,
            ELEMENT
        }

        virtual Boolean
        register_context(
            const QName& context_name,
            const QName& context_type,
            ContextType type = TYPE,
            Boolean  is_header = false
        ) = 0;
        ...
    };
};
```

The preceding `IT_Bus::ContextRegistry::register_context()` function takes the following arguments:

- `context_name`—the context name identifies the registered context. The context names for the pre-registered contexts are given in "Getting and Setting Transport Attributes" on page 206.
- `context_type`—the qualified name of the context data type or element. which can be either of the following:
    - The name of a schema type (that is, any type derived from `xsd:anyType`), or
    - The name of a schema element.

- `type`—a flag that indicates whether the `context_type` parameter is the name of a schema type (indicated by `IT_Bus::ContextRegistry::TYPE`) or the name of a schema element (indicated by `IT_Bus::ContextRegistry::ELEMENT`).
- `is_header`—for registering regular contexts (not headers), this flag should *not* be supplied (defaults to `false`).

Registering a non-serializable context

A non-serializable context can be any C++ type (that is, not necessarily inheriting from `IT_Bus::AnyType`). Example 80 shows the signature of the `register_context_data()` function in the `IT_Bus::ContextRegistry` class, which is used to register a non-serializable context.

Example 80: *The register_context_data() Function for Non-Serializable Contexts*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API ContextRegistry
    {
      public:
        virtual Boolean
        register_context_data(
            const QName& context_name
        ) = 0;
        ...
    };
};
```

The preceding `IT_Bus::ContextRegistry::register_context_data()` function takes the following argument:

- `context_name`—the name of a non-serializable context.

Registering header contexts

You can register the following kinds of header context:

- Registering a SOAP header context.
- Registering a CORBA header context.

Registering a SOAP header context

Example 81 shows the signature of the `register_context()` function and the `register_context_as_element()` function in the `IT_Bus::ContextRegistry` class, which are used to register a header context data type for the SOAP protocol.

Example 81: *The register_context() Function for SOAP Contexts*

```
// C++
namespace IT_Bus {
    class IT_BUS_API ContextRegistry
```

Example 81: *The register_context() Function for SOAP Contexts*

```
    {
      public:
        virtual Boolean
        register_context(
            const QName&  context_name,
            const QName&  context_type,
            const QName&  message_name,
            const String& part_name
        ) = 0;

        virtual Boolean
        register_context_as_element(
            const QName& context_name,
            const QName& element_name,
            const QName& message_name,
            const String& part_name
        ) = 0;
        ...
    };
};
```

The `IT_BUS::ContextRegistry::register_context()` function takes the following arguments:

- `context_name`—the context name identifies the registered context. A context name is needed, because a context type could be registered more than once (for example, if the same context type was used with different protocols).
- `context_type`—the qualified name of the context data type. It can be any schema type (that is, any type derived from `xsd:anyType`).
- `message_name`—this value corresponds to the `message` attribute in a `soap:header` element. Currently, the message name is ignored, but it should not clash with any existing message names.
- `part_name`—this value corresponds to the `part` attribute in a `soap:header` element. Currently, the part name is ignored.

The `IT_BUS::ContextRegistry::register_context_as_element()` function is a variant that enables you to base the context data on a specified XML element, `element_name`, rather than on a particular XML type.

Registering a CORBA header context

Example 82 shows the signature of the `register_context()` function in the `IT_Bus::ContextRegistry` class, which is used to register a context data type with the CORBA context container.

Example 82: *The register_context() Function for CORBA Contexts*

```
// C++
namespace IT_Bus {
    class IT_BUS_API ContextRegistry
    {
      public:
```

Example 82: *The register_context() Function for CORBA Contexts*

```
        virtual Boolean
        register_context(
            const QName&         context_name,
            const QName&         context_type
            const unsigned long  context_id,
        ) = 0;
    };
};
```

The `IT_Bus::ContextRegistry::register_context()` function takes the following arguments:

- `context_name`—the context name identifies the registered context. A context name is needed, because a context type could be registered more than once (for example, if the same context type was used with different protocols).
- `context_type`—the qualified name of the context data type. It can be any schema type (that is, any type derived from `xsd:anyType`).
- `context_id`—an ID that tags the GIOP service context containing the Artix context. In CORBA, the `context_id` corresponds to a service context ID of `IOP::ServiceId` type. For details of GIOP service contexts, consult the OMG CORBA specification.

> **Note:** Care should be exercised to avoid clashing with standard IDs allocated by the OMG, which are reserved for use either by the OMG itself or by particular ORB vendors. In particular, IDs in the range 0–4095 are reserved for use by the OMG.

Reading and Writing Context Data

You can read and write a variety of different kinds of context data: basic types, user-defined types, and instances of arbitrary C++ classes (custom types). This section describes how to access and modify the various kinds of context data.

Getting a Context Instance

Figure 25 shows an overview of how context data instances are accessed for writing and reading in an Artix application.

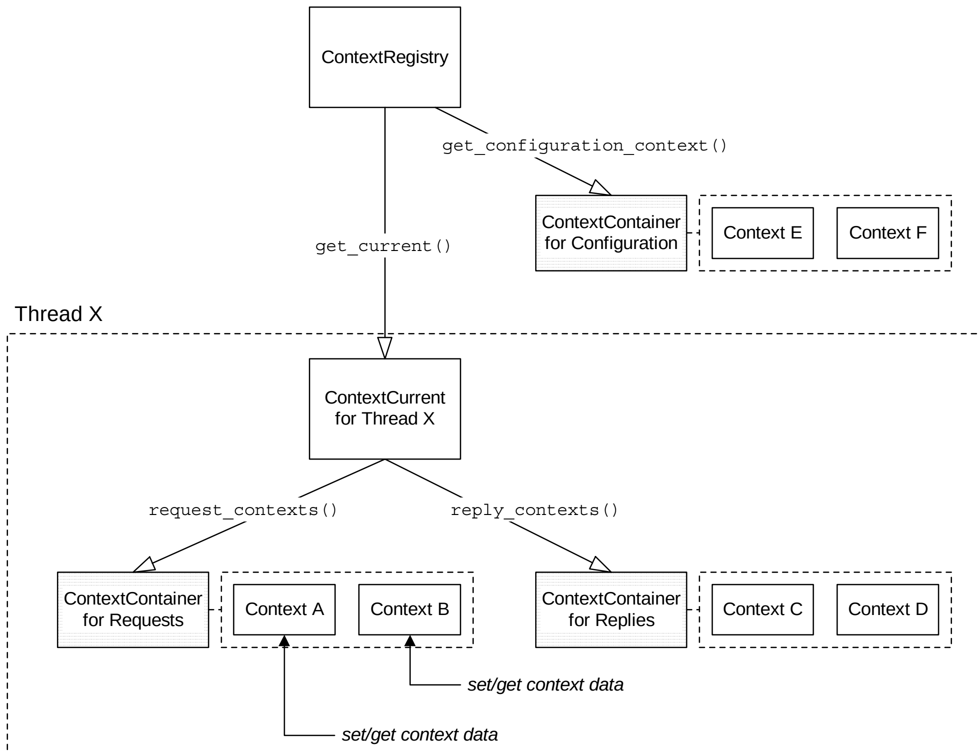

Figure 25: *Overview of Context Data and Context Containers*

Context containers

A *context container* is an object that holds a collection of contexts associated with a particular thread. There are three kinds of context container:

- *Request context container*—contains thread-specific context data that can be used for the following purposes:
  - Setting transport attributes on the client side that can be set *after* a connection has initialized,
  - Sending header contexts in outgoing request messages,
  - Receiving header contexts from incoming request messages.
- *Reply context container*—contains thread-specific context data that can be used for the following purposes:
  - Setting transport attributes on the server side that can be set *after* a connection has initialized,
  - Sending header contexts in outgoing reply messages,
  - Receiving header contexts from incoming reply messages.

- *Configuration context container*—contains endpoint-specific (but thread-independent) context data that can be set *before* a connection has initialized. Currently, Artix supports just the following configuration context properties:
  - HTTP endpoint URL,
  - JMS broker connection security information,
  - FTP connection settings.

Getting a configuration context container

To get a pointer to a configuration context container, call the `get_configuration_container()` function on the `ContextRegistry`, as shown in Example 83. The configuration context container is *endpoint-specific*, so you must specify the service QName, `service_name`, and the port name, `port_name`, of the relevant endpoint. Only the proxies and the servant objects associated with the specified endpoint are affected by the settings in this configuration context container.

Example 83: *Getting a Configuration ContextContainer Instance*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API ContextRegistry
    {
        virtual ContextContainer *
        get_configuration_context(
            const QName &  service_name,
            const String & port_name,
            bool           create_if_not_found = false
        ) = 0;
        ...
    };
};
```

Getting a ContextCurrent instance

To get a reference to a context registry instance, call the `IT_Bus::ContextRegistry::get_current()` function, as defined in Example 84.

Example 84: *Getting a ContextCurrent Instance*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API ContextRegistry
    {
        virtual ContextCurrent& get_current() = 0;
        ...
    };
};
```

ContextCurrent class

A *context current* is an object that holds references to thread-specific context data. In particular, an `IT_Bus::ContextCurrent` instance provides access to request contexts (through an `IT_Bus::ContextContainer` object) and reply contexts (through an `IT_Bus::ContextContainer` object).

Example 85 shows the declaration of the `IT_Bus::ContextCurrent` class, which defines two functions: `request_contexts()`, which returns a reference to the request context container, and `reply_contexts()`, which returns a reference to the reply context container.

Example 85: *The IT_Bus::ContextCurrent Class*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API ContextCurrent
    {
      public:

        virtual ContextContainer*
        request_contexts() = 0;

        virtual ContextContainer*
        reply_contexts() = 0;
    };
}
```

ContextContainer class

Example 86 shows the declaration of the `IT_Bus::ContextContainer` class, which defines member functions for getting and setting context objects.

Example 86: *The IT_Bus::ContextContainer Class*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API ContextContainer
    {
      public:
        // Get a serializable context
        virtual AnyType*
        get_context(
            const QName& context_name,
            bool         create_if_not_found = false
        ) = 0;

        virtual const AnyType*
        get_context(
            const QName& context_name
        ) const = 0;

        // Add a serializable context
```

Example 86: *The IT_Bus::ContextContainer Class*

```
        virtual Boolean
        add_context(
            const QName& context_name,
            AnyType&     context
        ) = 0;

        // Get a non-serializable context.
        virtual Context*
        get_context_data(const QName& context_name) = 0;

        virtual const Context*
        get_context_data(const QName& context_name) const = 0;

        // Add a non-serializable context.
        virtual Boolean
        add_context(
            const QName& context_name,
            Context&     context
        ) = 0;

        // Miscellaneous context functions
        virtual bool
        contains(const QName& context_name) = 0;

        virtual Boolean
        remove_context(const QName& context_name) = 0;
        ...
    };
};
```

Accessing and modifying serializable contexts

The `ContextContainer` class defines the following member functions for accessing and modifying serializable contexts:

- `get_context()`—returns a pointer to the context with the specified context name, `context_name`, which must have been previously registered with the context registry. The returned reference can be used either to read to or write from a context. The `create_if_not_found` flag has the following effect:
    - If `false` and the context is not found, the returned pointer value is `NULL`.
    - If `true` and the context is not found, the return value points at a newly created context instance.
- `add_context()`—is a convenience function that lets you set a context from an existing context instance. The context must have been previously registered with the context registry.

Accessing and modifying non-serializable contexts

The ContextContainer class defines the following member functions for accessing and modifying non-serializable contexts:

- get_context_data()—returns a pointer to the context with the specified context name, context_name, which must have been previously registered with the context registry. The returned reference can be used either to read to or write from a context.
- add_context()—is a convenience function that lets you set a context from an existing context instance. The context parameter must be defined as an IT_Bus::ContextT<*DataType*> type, which is used to wrap an instance of *DataType*.

Reading and Writing Basic Types

To insert and extract a basic type, *BasicType*, you must use its corresponding *BasicType*Holder type. For example, to insert an IT_Bus::String type into a context, you must first insert the string into an IT_Bus::StringHolder object. This approach is necessary because the get_context() and add_context() functions expect context data to be a type that derives from IT_Bus::AnyType.

For a complete list of Holder types, see "Holder Types" on page 307.

Registering a context for strings

For example, to register a *configuration context* that holds string data, you could use code like the following:

```
// C++
const IT_Bus::QName test_ctx_name(
    "", "TestString", "http://www.iona.com/test/context"
);

reg->register_context(
    test_ctx_name,
    IT_Bus::StringHolder().get_type()
};
```

Where reg is a context registry (of IT_Bus::ContextRegistry type). The IT_Bus::StringHolder() constructor creates a temporary instance of a StringHolder object, which you can use to get the QName of the StringHolder type.

Inserting a basic type into a context

The following example shows how to insert an `IT_Bus::StringHolder` instance into the `test_ctx_name` request context.

```
// C++
IT_Bus::AnyType* any_string = request_contexts->get_context(
    test_ctx_name,    // The name of the string context.
    true              // The create_if_not_found flag
);

IT_Bus::StringHolder* str_holder =
   dynamic_cast<IT_Bus::StringHolder*>(any_string);

str_holder->set("Hello World!");
```

Extracting a basic type from a context

The following example shows how to extract the `IT_Bus::StringHolder` instance from the `test_ctx_name` request context.

```
// C++
IT_Bus::AnyType* any_string = request_contexts->get_context(
    test_ctx_name     // The name of the string context.
);

IT_Bus::StringHolder* str_holder =
   dynamic_cast<IT_Bus::StringHolder*>(any_string);

IT_Bus::String str = str_holder->get();
```

Reading and Writing User-Defined Types

You can define a dedicated user-defined schema type to hold the context data. You could include the context type definition directly in the application's WSDL contract; however, it is usually more convenient to define the context type in a separate XML schema file.

For example, to define a complex context data type, *ContextDataType*, in the namespace, *ContextDataURI*, you could define a context schema following the outline shown in Example 87.

Example 87: *Outline of a Context Schema*

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="ContextDataURI"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

    <xs:complexType name="ContextDataType">
        ...
```

Example 87: *Outline of a Context Schema*

```
    </xs:complexType>

</xs:schema>
```

Generating stubs from a context schema

To generate C++ stubs from a context schema file, *ContextSchema*.xsd, enter the following command at the command line:

wsdltocpp *ContextSchema*.xsd

The WSDL-to-C++ compiler generates the following C++ stub files:

*ContextSchema*_wsdlTypes.h
*ContextSchema*_wsdlTypesFactory.h
*ContextSchema*_wsdlTypes.cxx
*ContextSchema*_wsdlTypesFactory.cxx

Registering a context for a user-defined type

For example, to register a *configuration context* that holds an instance of the *ContextDataType* type, you could use code like the following:

```
// C++
const IT_Bus::QName userdata_ctx_name(
    "", "TestUserData", "http://www.iona.com/test/context"
);
const IT_Bus::QName userdata_ctx_type(
    "", "ContextDataType", "ContextDataURI"
);

reg->register_context(
    userdata_ctx_name,
    userdata_ctx_type
);
```

Where reg is a context registry (of IT_Bus::ContextRegistry type).

Inserting a user-defined type into a context

The following example shows how to insert a *ContextDataType* instance into the `userdata_ctx_name` request context.

```
// C++
IT_Bus::AnyType* any_userdata = request_contexts->get_context(
    userdata_ctx_name,    // The name of the UserData context.
    true                  // The create_if_not_found flag
);

ContextDataType* ctx_data =
   dynamic_cast<ContextDataType*>(any_userdata);
ctx_data->set...()        // Initialize the context data.
```

Extracting a user-defined type from a context

The following example shows how to extract the *ContextDataType* instance from the `userdata_ctx_name` request context.

```
// C++
IT_Bus::AnyType* any_userdata = request_contexts->get_context(
    userdata_ctx_name     // The name of the UserData context.
);

ContextDataType* ctx_data =
   dynamic_cast<ContextDataType*>(any_userdata);
cout << ctx_data->get...()      // Initialize the context data.
```

Reading and Writing Custom Types

Sometimes it is necessary to store a custom data type in a context—that is, a data type that does not inherit from `IT_Bus::AnyType`. Using a non-serializable context, you can store instances of *any* class in a context.

Note: Non-serializable contexts are not streamable, however. You can only set and get this kind of context locally, from within the same process.

ContextT template

The ContextT<T> template class is used to hold a reference to an arbitrary C++ type. The ContextT<T> type is needed to wrap T instances before they can be added to a context container.

Example 88: *The ContextT Template Class*

```
// C++
namespace IT_Bus {
    template<class T>
    class ContextT : public Context
    {
      public:
        ContextT(T& context) : m_context(context)
        {
            // complete
        }

        T& get_data()  {
            return m_context;
        }

      private:
        T& m_context;
    };
};
```

Inserting a custom type into a context

Given a user-defined type, CustomClass, and a registered custom context name, CUSTOM_CTX_NAME, the following example shows how to use the ContextT<> template to store a CustomClass instance in a request context container.

```
// C++
using namespace IT_Bus;

typedef ContextT<CustomClass> CustomClassContext;

CustomClass data;
CustomClassContext ctx(data);
request_contexts->add_context(CUSTOM_CTX_NAME, ctx);
```

Extracting a custom type from a context

The following example shows how to extract a `CustomClass` instance from the request context container. The code that extracts the context must be colocated with the code that inserts it (in other words, this type of context *cannot* be sent in a header).

```
// C++
using namespace IT_Bus;

typedef ContextT<CustomClass> CustomClassContext;

Context * ctx =
   request_contexts->get_context_data(CUSTOM_CTX_NAME);
CustomClassContext* custom_ctx =
   dynamic_cast<CustomClassContext*>(result_ctx);
CustomClass& custom = custom_ctx->get_data();
```

Accessing the server operation context

For a practical application of non-serializable contexts, consider Example 89 which shows you how to access an `IT_Bus::ServerOperation` instance in the context of an invocation on the server side (in other words, this code could appear in the body of a servant function).

Example 89: *Accessing the Server Operation Context*

```
// C++
#include <it_bus_pdk/context.h>
#include <it_bus_pdk/context_attrs/context_constants.h>
#include <it_bus/operation.h>

using namespace IT_Bus;
using namespace IT_ContextAttributes;

ContextRegistry* context_registry =
    bus->get_context_registry();

// Obtain a reference to the ContextCurrent.
ContextCurrent& context_current =
    context_registry->get_current();

// Obtain a pointer to the RequestContextContainer.
ContextContainer* context_container =
    context_current.request_contexts();

ServerOperation *  operation = 0;

// Users can now access context derived from Context class.
Context* context_data =
 context_container->get_context_data(SERVER_OPERATION_CONTEXT);

// Need to cast to appropriate context type.
ServerOperationContext* operation =
      dynamic_cast<ServerOperationContext*>(context_data);
```

Example 89: *Accessing the Server Operation Context*

```
// ServerOperation is wrapped in a template ContextT class.
ServerOperation& server_op = operation->get_data();
```

Durability of Context Settings

When you set a context value using either `get_context()` or `add_context()`, the context value is not valid indefinitely. The general rule is that a context value is valid only for the duration of an invocation. There are two cases two consider, as follows:

- Client side durability.
- Server side durability.

Client side durability

On the client side, the general rule is that a context value affects only the next invocation in the current thread. At the end of the invocation, Artix clears the context value. Hence, it is generally necessary to reset the context value before the making the next invocation.

An exception to this rule is demonstrated by the context types derived from the `http-conf` schema (`HTTP_CLIENT_OUTGOING_CONTEXTS` and `HTTP_CLIENT_INCOMING_CONTEXTS`). These context values are valid over multiple invocations from the current thread.

Server side durability

On the server side, the general rule is that context values are set at the start of an operation invocation (when the server receives a request message) and cleared at the end of the invocation. Context values are thus available to the servant code *only* for the duration of the invocation.

An exception to this rule is the value of an endpoint URL, which can be modified outside of an invocation context by calling the `setURL()` function on a server configuration context. For details of how to do this, see "Setting a Configuration Context on the Server Side" on page 176.

Context Example

This section shows how to modify the settings in a context, using the `http-conf` schema as an example. The `http-conf:clientType` context type enables you to modify the client port settings on a HTTP port and the `http-conf:serverType` context type enables you to modify server endpoint settings.

This section contains:

- HTTP-Conf Schema
- Setting a Request Context on the Client Side
- Setting a Configuration Context on the Server Side

HTTP-Conf Schema

This subsection provides an overview of the `http-conf` schema, which provides the definitions of the `http-conf` configuration context types. Using the `http-conf` schema, you can configure the properties of a HTTP port either in a WSDL contract or by programming. The C++ mapping of the `http-conf` contexts are already generated for you—all that you need to do is include the relevant header file in your code and link with the relevant library.

http-conf schema file

The `http-conf` schema defines WSDL extension elements for configuring a HTTP port in Artix. The `http-conf` schema is defined in the following file:

ArtixInstallDir/`cxx_java/schemas/http-conf.xsd`

http-conf:clientType XML definition

Example 90 gives an extract from the `http-conf` schema, showing part of the definition of the `http-conf:clientType` complex type.

Example 90: *Definition of the http-conf:clientType Type*

```
<xs:schema
   targetNamespace="http://schemas.iona.com/transports/http/configuration"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"
           xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">

    <xs:import namespace="http://schemas.xmlsoap.org/wsdl/"/>
    ...
    <xs:complexType name="clientType">
        <xs:complexContent>
            <xs:extension base="wsdl:tExtensibilityElement">
                <xs:attribute name="SendTimeout"
                              type="http-conf:timeIntervalType"
                              use="optional" default="30000"/>

                <xs:attribute name="ReceiveTimeout"
                              type="http-conf:timeIntervalType"
                              use="optional"
                              default="30000"/>
                ...
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    ...
</xs:schema>
```

http-conf timeout attributes

The `http-conf:clientType` type defines two timeout attributes, as follows:

- `SendTimeout`—(in milliseconds) the maximum amount of time a client will spend attempting to contact a remote server.
- `ReceiveTimeout`—(in milliseconds) for synchronous calls, the maximum amount of time a client will wait for a server response.

http-conf:clientType C++ mapping

The `http-conf:clientType` port type maps to the `IT_ContextAttributes::clientType` C++ class, as shown in Example 91. The `SendTimeout` and `ReceiveTimeout` attributes each map to get and set functions. Because these are optional attributes, the get functions return a pointer. A `NULL` return value indicates that the attribute is not set.

Example 91: *C++ Mapping of http-conf:clientType Type*

```
// C++
...
namespace IT_ContextAttributes
{
    class clientType
      : public IT_tExtensibilityElementData,
        public virtual IT_Bus::ComplexContentComplexType
    {
      public:
        ...
        IT_Bus::Int *       getSendTimeout();
        const IT_Bus::Int * getSendTimeout() const;
        void setSendTimeout(const IT_Bus::Int * val);
        void setSendTimeout(const IT_Bus::Int & val);

        IT_Bus::Int *       getReceiveTimeout();
        const IT_Bus::Int * getReceiveTimeout() const;
        void setReceiveTimeout(const IT_Bus::Int * val);
        void setReceiveTimeout(const IT_Bus::Int & val);
        ...
    };
};
```

http-conf:serverType C++ mapping

The `http-conf:serverType` port type maps to the `IT_ContextAttributes::serverType` C++ class, as shown in Example 92.

In this example, we are only interested in the functions for setting and getting the endpoint URL, `setURL()` and `getURL()`. Using these functions, you can examine or modify the host and IP port where the server listens for incoming client connections.

Example 92: *C++ Mapping of the http-conf:serverType Type*

```
// C++
...
namespace IT_ContextAttributes {
    class IT_CONTEXT_ATTRIBUTE_API serverType
     : public IT_tExtensibilityElementData,
       public virtual IT_Bus::ComplexContentComplexType
    {
      public:
        ...
        IT_Bus::String * getURL();
        const IT_Bus::String * getURL() const;
        void setURL(const IT_Bus::String * val);
        void setURL(const IT_Bus::String & val);
        ...
    };
};
```

Header and library files

One of the pre-requisites for programmatically modifying the `http-conf` port configuration is to include the following header file in your C++ code:

`it_bus_pdk/context_attrs/http_conf_xsdTypes.h`

You must also link your client application with the following library file:

Windows

ArtixInstallDir`/lib/it_context_attribute.lib`

UNIX

ArtixInstallDir`/lib/it_context_attribute.so`
ArtixInstallDir`/lib/it_context_attribute.sl`
ArtixInstallDir`/lib/it_context_attribute.a`

Pre-registered context type names

The `http-conf:clientType` context type for outgoing data is pre-registered with the context registry under the following QName constant:

`IT_ContextAttributes::HTTP_CLIENT_OUTGOING_CONTEXTS`

The `http-conf:serverType` context type for outgoing data is pre-registered with the context registry under the following QName constant:

`IT_ContextAttributes::HTTP_SERVER_OUTGOING_CONTEXTS`

Setting a Request Context on the Client Side

This subsection describes how to set attributes on the `http-conf:clientType` context (corresponds to the attributes settable on the `<http-conf:client>` WSDL port extensor). The `http-conf:clientType` context configures client-side attributes on the HTTP transport plug-in.

Client main function

Example 93 shows sample code from a client main function, which shows how to initialize `http-conf:clientType` context data in the current thread.

Example 93: *Client Main Function Setting a Request Context*

```
// C++

#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_cal/iostream.h>

// Include header files related to the soap context
1 #include <it_bus_pdk/context.h>
2 #include <it_bus_pdk/context_attrs/http_conf_xsdTypes.h>

IT_USING_NAMESPACE_STD

using namespace IT_ContextAttributes;
using namespace IT_Bus;

int
main(int argc, char* argv[])
{
    try
    {
        IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);

3       ContextRegistry* context_registry =
            bus->get_context_registry();

        // Obtain a reference to the ContextCurrent
4       ContextCurrent& context_current =
            context_registry->get_current();

        // Obtain a pointer to the Request ContextContainer
5       ContextContainer* context_container =
            context_current.request_contexts();

        // Obtain a reference to the context
6       AnyType* info = context_container->get_context(
           IT_ContextAttributes::HTTP_CLIENT_OUTGOING_CONTEXTS,
            true
        );

        // Cast the context into a clientType object
7       clientType* http_client_config =
            dynamic_cast<clientType*> (info);
```

Example 93: *Client Main Function Setting a Request Context*

8
```
        // Modify the Send/Receive timeouts
        http_client_config->setSendTimeout(2000);
        http_client_config->setReceiveTimeout(600000);
        ...
    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Error : Unexpected error occured!"
             << endl << e.message()
             << endl;
        return -1;
    }
    return 0;
}
```

The preceding code example can be explained as follows:

1. The `it_bus_pdk/context.h` header file contains the declarations of the following classes:
   - `IT_Bus::ContextRegistry`,
   - `IT_Bus::ContextContainer`,
   - `IT_Bus::ContextCurrent`.
2. The `http_conf_xsdTypes.h` header declares the context data types generated from the `http-conf` schema.
3. Obtain a reference to the `IT_Bus::ContextRegistry` object, which is used to register contexts with the Bus.
4. Call `IT_Bus::ContextRegistry::get_current()` to obtain a reference to the `IT_Bus::ContextCurrent` object. The current object provides access to the context objects associated with the current thread.
5. Call `IT_Bus::ContextContainer::request_contexts()` to obtain an `IT_Bus::ContextContainer` object that contains all of the contexts for requests originating from the current thread.
6. The `IT_Bus::ContextContainer::get_context()` function is called with its second parameter set to `true`, indicating that a context with that name should be created if none already exists.
7. The `IT_Bus::AnyType` class is the base type for all complex types in Artix. In this case, you can cast the `AnyType` instance, `info`, to its derived type, `clientType`.
8. You can now modify the send and receive timeouts on the client port using `setSendTimeout()` and `setReceiveTimeout()`. These timeouts will be applied to any subsequent calls issuing from the current thread.

Setting a Configuration Context on the Server Side

This subsection describes how to set attributes on the `http-conf:serverType` context (corresponds to the attributes settable on the `<http-conf:server>` WSDL port extensor). The `http-conf:serverType` context configures server-side attributes on the HTTP transport plug-in.

Server main function

Example 94 shows sample code from a server main function, which shows how to initialize `http-conf:serverType` configuration context data.

Example 94: *Server Main Function Setting a Configuration Context*

```
// C++

#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_cal/iostream.h>

// Include header files related to the soap context
1 #include <it_bus_pdk/context.h>
2 #include <it_bus_pdk/context_attrs/http_conf_xsdTypes.h>

IT_USING_NAMESPACE_STD

using namespace IT_ContextAttributes;
using namespace IT_Bus;

int
main(int argc, char* argv[])
{
    try
    {
        IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);

3       IT_Bus::QName service_name(
            "",
            "SOAPService",
        "http://www.iona.com/hello_world_soap_http"
        );

4       ContextRegistry* context_registry =
            bus->get_context_registry();

5       ContextContainer * context_container =
            context_registry->get_configuration_context(
              service_name,
              "SoapPort",
              true
           );

        // Obtain a reference to the context
6       AnyType* info = context_container->get_context(
           IT_ContextAttributes::HTTP_SERVER_OUTGOING_CONTEXTS,
            true
        );

        // Cast the context into a serverType object
7       serverType* http_server_config =
            dynamic_cast<serverType*> (info);

        // Modify the endpoint URL
8       http_server_config->setURL("http://localhost:63278");
        ...
```

Example 94: *Server Main Function Setting a Configuration Context*

```
        GreeterImpl servant(bus);
        bus->register_servant(
                    servant,
                    "../../etc/hello_world.wsdl",
                    service_name
             );
    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Error : Unexpected error occured!"
             << endl << e.message()
             << endl;
        return -1;
    }
    return 0;
}
```

The preceding code example can be explained as follows:

1. The `it_bus_pdk/context.h` header file contains the declarations of the following classes:
   - `IT_Bus::ContextRegistry`,
   - `IT_Bus::ContextContainer`,
   - `IT_Bus::ContextCurrent`.
2. The `http_conf_xsdTypes.h` header declares the context data types generated from the `http-conf` schema.
3. This `service_name` is the QName of the SOAP service featured in the `hello_world_soap_http` demonstration (in `samples/basic/hello_world_soap_http`).
4. Obtain a reference to the `IT_Bus::ContextRegistry` object, which is used to register contexts with the Bus.
5. The `IT_Bus::ContextContainer` object returned by `get_configuration_context()` holds configuration data that is used exclusively by the specified endpoint (that is, the `SoapPort` port in the `SOAPService` service).
6. The `IT_Bus::ContextContainer::get_context()` function is called with its second parameter set to `true`, indicating that a context with that name should be created if none already exists.
7. The `IT_Bus::AnyType` class is the base type for all complex types in Artix. In this case, you can cast the `AnyType` instance, `info`, to its derived type, `serverType`.
8. You can now modify the URL used by the `SoapPort` port by calling the `setURL()` function.

SOAP Header Contexts

This section provides a detailed discussion of the custom SOAP header demonstration, which shows you how to propagate context data in a SOAP header.

This section contains:

Custom SOAP Header Demonstration

The examples in this section are based on the custom SOAP header demonstration, which is located in the following Artix directory:

ArtixInstallDir/samples/advanced/custom_soap_header

Figure 26 shows an overview of the custom SOAP header demonstration, showing how the client piggybacks context data along with an invocation request that is invoked on the sayHi operation.

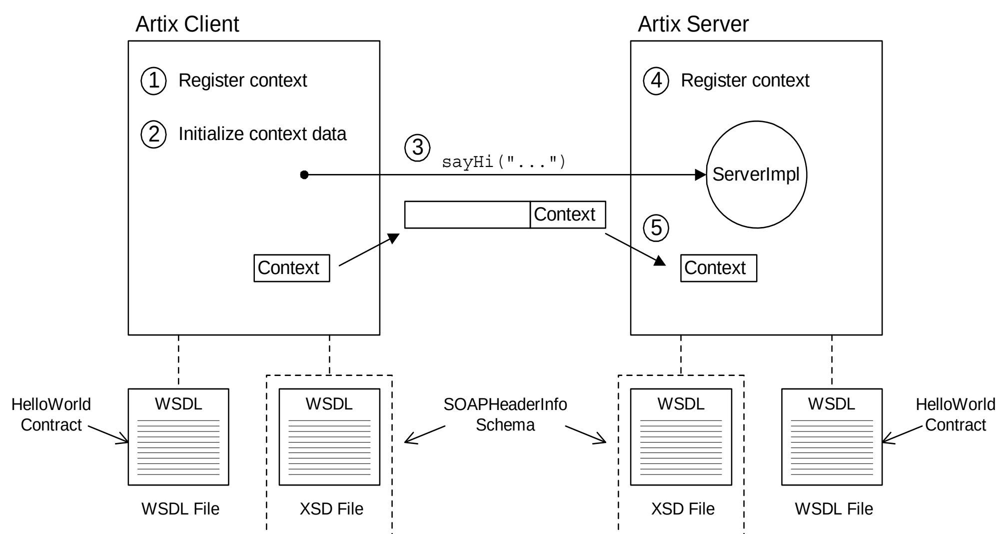

Figure 26: *Overview of the Custom SOAP Header Demonstration*

Transmission of context data

As illustrated in Figure 26, SOAP context data is transmitted as follows:

1. The client registers the context type, SOAPHeaderInfo, with the Bus.
2. The client initializes the context data instance.
3. The client invokes the sayHi() operation on the server.
4. As the server starts up, it registers the SOAPHeaderInfo context type with the Bus.
5. When the sayHi() operation request arrives on the server side, the sayHi() operation implementation extracts the context data from the request.

HelloWorld WSDL contract

The HelloWorld WSDL contract defines the contract implemented by the server in this demonstration. In particular, the HelloWorld contract defines the `Greeter` port type containing the `sayHi` WSDL operation.

SOAPHeaderInfo schema

The `SOAPHeaderInfo` schema (in the `samples/advanced/custom_soap_header/etc/contextTypes.xsd` file) defines the custom data type used as the context data type. This schema is specific to the custom SOAP header demonstration.

SOAP Header Context Schema

This subsection describes how to define an XML schema for a context type. In this example, the `SOAPHeaderInfo` type is declared in an XML schema. The `SOAPHeaderInfo` type is then used by the custom SOAP header demonstration to send custom data in a SOAP header.

SOAPHeaderInfo XML declaration

Example 95 shows the schema for the `SOAPHeaderInfo` type, which is defined specifically for the custom SOAP header demonstration to carry some sample data in a SOAP header. Note that Example 95 is a pure schema declaration, *not* a WSDL declaration.

Example 95: *XML Schema for the SOAPHeaderInfo Context Type*

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://schemas.iona.com/types/context"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:complexType name="SOAPHeaderInfo">
        <xs:annotation>
            <xs:documentation>
                Content to be added to a SOAP header
            </xs:documentation>
        </xs:annotation>
        <xs:sequence>
            <xs:element name="originator" type="xs:string"/>
            <xs:element name="message" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

The `SOAPHeaderInfo` complex type defines two member elements, as follows:

- `originator`—holds an arbitrary client identifier.
- `message`—holds an arbitrary example message.

Target namespace

You can use any target namespace for a context schema (as long as it does not clash with an existing namespace). This demonstration uses the following target namespace:

```
http://schemas.iona.com/types/context
```

Compiling the SOAPHeaderInfo schema

To compile the `SOAPHeaderInfo` schema, invoke the `wsdltocpp` compiler utility at the command line, as follows:

```
wsdltocpp contextTypes.xsd
```

Where `contextTypes.xsd` is a file containing the XML schema from Example 95. This command generates the following C++ stub files:

```
contextTypes_xsdTypes.h
contextTypes_xsdTypesFactory.h
contextTypes_xsdTypes.cxx
contextTypes_xsdTypesFactory.cxx
```

SOAPHeaderInfo C++ mapping

Example 96 shows how the schema from Example 95 on page 180 maps to C++, to give the `soap_interceptor::SOAPHeaderInfo` C++ class.

Example 96: *C++ Mapping of the SOAPHeaderInfo Context Type*

```
// C++
...
namespace soap_interceptor
{
    ...
    class SOAPHeaderInfo : public IT_Bus::SequenceComplexType
    {
      public:
        static const IT_Bus::QName type_name;

        SOAPHeaderInfo();
        SOAPHeaderInfo(const SOAPHeaderInfo & copy);
        virtual ~SOAPHeaderInfo();
        ...
        IT_Bus::String &      getoriginator();
        const IT_Bus::String & getoriginator() const;
        void setoriginator(const IT_Bus::String & val);

        IT_Bus::String &      getmessage();
        const IT_Bus::String & getmessage() const;
        void setmessage(const IT_Bus::String & val);
        ...
    };
    ...
}
```

Declaring the SOAP Header Explicitly

There are two different approaches you can take with SOAP headers:

- *Implicit SOAP header*—(the approach taken in Example 95 on page 180) in this case, you need only declare the schema type that holds the header data. By registering the type as a SOAP header context, you enable an Artix application to send and receive SOAP headers of this type.
- *Explicit SOAP header*—in this case, you must modify the original WSDL contract and explicitly declare which operations can send and receive the header. This approach might be useful for certain interoperability scenarios.

This subsection briefly describes how to implement the second approach, explicitly declaring the SOAP header.

> **Note:** The implicit approach is also consistent with the SOAP specification, which does *not* require you to declare SOAP headers explicitly in WSDL.

Demonstration code

The code for this demonstration is located in the following directory:

ArtixInstallDir/cxx_java/samples/advanced/soap_header_binding

SOAP header declaration

Example 97 shows how to declare a SOAP header, of SOAPHeaderData type, explicitly in a WSDL contract.

Example 97: *SOAP Header Declared in the WSDL Contract*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloWorld"
    targetNamespace="http://www.iona.com/soap_header"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:http-conf="http://schemas.iona.com/transports/http/configu
   ration"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.iona.com/soap_header"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <types>
  <schema targetNamespace="http://www.iona.com/soap_header"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
            <element name="responseType" type="xsd:string"/>
            <element name="requestType" type="xsd:string"/>
1           <complexType name="SOAPHeaderData">
                <sequence>
                    <element name="originator" type="xsd:string"/>
                    <element name="message" type="xsd:string"/>
                </sequence>
```

Example 97: *SOAP Header Declared in the WSDL Contract*

```
            </complexType>
2           <element name="SOAPHeaderInfo"
                    type="tns:SOAPHeaderData"/>
        </schema>
    </types>

    <message name="sayHiRequest"/>
    <message name="sayHiResponse">
        <part element="tns:responseType" name="theResponse"/>
    </message>
    ...
3   <message name="header_message">
        <part element="tns:SOAPHeaderInfo" name="header_info"/>
    </message>
    <portType name="Greeter">
        <operation name="sayHi">
          <input message="tns:sayHiRequest" name="sayHiRequest"/>
            <output message="tns:sayHiResponse"
                    name="sayHiResponse"/>
        </operation>
        ...
    </portType>

    <binding name="Greeter_SOAPBinding" type="tns:Greeter">
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="sayHi">
            <soap:operation soapAction="" style="document"/>
            <input name="sayHiRequest">
                <soap:body use="literal"/>
4             <soap:header message="tns:header_message"
                           part="header_info"
                           use="literal"/>
            </input>
            <output name="sayHiResponse">
                <soap:body use="literal"/>
              <soap:header message="tns:header_message"
                           part="header_info"
                           use="literal"/>
            </output>
        </operation>
        ...
    </binding>
    ...
</definitions>
```

The preceding WSDL contract can be explained as follows:

1. This example declares a header of type `SOAPHeaderData` (this example is different from the header type declared in Example 95 on page 180). The `SOAPHeaderData` type contains two string fields, `originator` and `message`.
2. You must declare an element to contain the header data. In this case, the header is transmitted as `<SOAPHeaderInfo> ... </SOAPHeaderInfo>`.

3. You must declare a `message` element for the header. In this case, the message QName is `tns:header_message` and the part name is `header_info`. These correspond to the values that would be passed to the last two arguments of the `IT_Bus::ContextRegistry::register_context()` function.
4. In the scope of the `binding` element, you should declare which operations include the `SOAPHeaderData` header, as shown. The `soap:header` element references the message QName, `tns:header_message`, and the part name, `header_info`.

Client Main Function

This subsection discusses the client for the custom SOAP header demonstration. This client is designed to send a custom header, of `SOAPHeaderInfo` type, every time it invokes an operation on the `Greeter` port type.

To enable the sending of context data, the client performs two fundamental tasks, as follows:

1. *Register a context type with the context registry*—registering the context type is a prerequisite for sending context data in a request. By registering the context type with the Bus, you give the Bus instance the capability to marshal and unmarshal context data of that type.
2. *Initialize the context data in the ContextCurrent object*—before invoking any operations, the client obtains an instance of the header context data from an `IT_Bus::ContextCurrent` object. After initializing the header context data, any operations invoked from the current thread will include the header context data.

Client main function

Example 98 shows sample code from the client main function, which shows how to register a context type and initialize header context data for the current thread.

Example 98: *Client Main Function Setting a SOAP Context*

```
// C++
// GreeterClientSample.cxx File

#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_cal/iostream.h>

// Include header files related to the soap context
1 #include <it_bus_pdk/context.h>

// Include header files representing the soap header content
2 #include "contextTypes_xsdTypes.h"
#include "contextTypes_xsdTypesFactory.h"

#include "GreeterClient.h"

IT_USING_NAMESPACE_STD
```

Example 98: *Client Main Function Setting a SOAP Context*

```
using namespace soap_interceptor;
using namespace IT_Bus;

int
main(int argc, char* argv[])
{
    try
    {
        IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);
        GreeterClient client;

3       ContextRegistry* context_registry =
            bus->get_context_registry();

        // Create QName objects needed to define a context
4       const QName principal_ctx_name(
            "",
            "SOAPHeaderInfo",
            ""
        );
5       const QName principal_ctx_type(
            "",
            "SOAPHeaderInfo",
            "http://schemas.iona.com/types/context"
        );
6       const QName principal_message_name(
            "soap_header",
            "header_content",
            "http://schemas.iona.com/custom_header"
        );
7       const String principal_part_name("header_info");

        // Register the context with the ContextRegistry
8       context_registry->register_context(
            principal_ctx_name,
            principal_ctx_type,
            principal_message_name,
            principal_part_name
        );

        // Obtain a reference to the ContextCurrent
9       ContextCurrent& context_current =
            context_registry->get_current();

        // Obtain a pointer to the RequestContextContainer
10      ContextContainer* context_container =
            context_current.request_contexts();

        // Obtain a reference to the context
11      AnyType* info = context_container->get_context(
            principal_ctx_name,
            true
        );

        // Cast the context into a SOAPHeaderInfo object
12      SOAPHeaderInfo* header_info =
            dynamic_cast<SOAPHeaderInfo*> (info);
```

Example 98: *Client Main Function Setting a SOAP Context*

```
        // Create the content to be added to the header
        const String originator(" Software");
        const String message("Artix is Powerful!");

        // Add the header content
        header_info->setoriginator(originator);
        header_info->setmessage(message);

        // Invoke the Web service business methods
        String theResponse;

13      client.sayHi(theResponse);
        cout << "sayHi response: " << theResponse << endl;
    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Error : Unexpected error occured!"
             << endl << e.message()
             << endl;
        return -1;
    }
    return 0;
}
```

The preceding code example can be explained as follows:

1. The `it_bus_pdk/context.h` header file contains the declarations of the following classes:
   - `IT_Bus::ContextRegistry`,
   - `IT_Bus::ContextContainer`,
   - `IT_Bus::ContextCurrent`.
2. The `contextTypes_xsdTypes.h` local header file contains the declaration of the `SOAPHeaderInfo` class, which has been generated from the context schema (see Example 95 on page 180).
3. Obtain a reference to the `IT_Bus::ContextRegistry` object, which is used to register contexts with the Bus.
4. The QName with local name, `SOAPHeaderInfo`, is a context name that identifies the context uniquely. Although the context name is specified as a QName, it does not refer to an XML element. You can choose any unique QName as the context name.
5. The QName with namespace URI, `http://schemas.iona.com/types/context`, and local part, `SOAPHeaderInfo`, identifies the context type from Example 95 on page 180.
6. The QName with namespace URI, `http://schemas.iona.com/custom_header`, and local part, `header_content`, corresponds to the `message` attribute of a `soap:header` element. The value is currently ignored (but should not clash with any existing message QNames).

7. The header_info string value identifies the part of the SOAP header that holds the context data. It corresponds to the part attribute of a soap:header element. The value is currently ignored.
8. The call to register_context() tells the Artix Bus that the SOAPHeaderInfo type will be used to send context data in SOAP headers. After you have registered the context, the Bus is prepared to marshal the context data (if any) into a SOAP header.
9. Call IT_Bus::ContextRegistry::get_current() to obtain a reference to the IT_Bus::ContextCurrent object. The current object provides access to all context objects associated with the current thread.
10. Call IT_Bus::ContextContainer::request_contexts() to obtain an IT_Bus::ContextContainer object that contains all of the contexts for requests originating from the current thread.
11. The IT_Bus::ContextContainer::get_context() function is called with its second parameter set to true, indicating that a context with that name should be created if none already exists.
12. The IT_Bus::AnyType class is the base type for all complex types in Artix. In this case, you can cast the AnyType instance, info, to its derived type, SOAPHeaderInfo.

    By setting the originator and message elements of this SOAPHeaderInfo object, you are effectively fixing the context data for all operations invoked from this thread.
13. When you invoke the sayHi() operation, the context data is included in the SOAP header. From this point on, any WSDL operation invoked from the current thread will include the SOAPHeaderInfo context data in its SOAP header.

Server Main Function

This subsection discusses the main function for the server in the custom SOAP header demonstration. In addition to the usual boilerplate code for an Artix server (that is, registering a servant and calling IT_Bus::run()), this server also registers a context type with the Bus.

By registering a context type with the Bus, you give the Bus instance the capability to unmarshal context data of that type. This unmarshalling capability is then exploited in the implementation of the sayHi() operation (see Example 100 on page 190).

Server main function

Example 99 shows sample code from the server main function, which registers the SOAPHeaderInfo context type and then creates and registers a GreeterImpl servant object.

Example 99: *Server Main Function Registering a SOAP Context*

```
// C++
#include <it_bus/bus.h>
```

Example 99: *Server Main Function Registering a SOAP Context*

```
#include <it_bus/exception.h>
#include <it_bus/fault_exception.h>
#include <it_cal/iostream.h>

1 #include <it_bus_pdk/context.h>

#include "GreeterImpl.h"

IT_USING_NAMESPACE_STD

using namespace soap_interceptor;
using namespace IT_Bus;

int
main(int argc, char* argv[])
{
    try
    {
        IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);

2       ContextRegistry* context_registry =
            bus->get_context_registry();

3       const QName principal_ctx_name(
            "",
            "SOAPHeaderInfo",
            ""
        );
4       const QName principal_ctx_type(
            "",
            "SOAPHeaderInfo",
            "http://schemas.iona.com/types/context"
        );
5       const QName principal_message_name(
            "soap_header",
            "header_content",
            "http://schemas.iona.com/custom_header"
        );
6       const String principal_part_name("header_info");

7       context_registry->register_context(
            principal_ctx_name,
            principal_ctx_type,
            principal_message_name,
            principal_part_name
        );

        GreeterImpl servant(bus);

        IT_Bus::QName service_name("", "SOAPService",
   "http://www.iona.com/custom_soap_interceptor");

        bus->register_servant(
            servant,
            "../../etc/hello_world.wsdl",
            service_name
        );
```

Example 99: *Server Main Function Registering a SOAP Context*

```
        IT_Bus::run();
    }
    catch(IT_Bus::Exception& e)
    {
        cout << "Error occurred: " << e.message() << endl;
        return -1;
    }
    return 0;
}
```

The preceding code example can be explained as follows:

1. The `it_bus_pdk/context.h` header file contains the declarations of the following classes:
    - `IT_Bus::ContextRegistry`,
    - `IT_Bus::ContextContainer`,
    - `IT_Bus::ContextCurrent`.
2. Obtain a reference to the `IT_Bus::ContextRegistry` object, which is used to register contexts with the Bus.
3. The QName with local name, `SOAPHeaderInfo`, is a context name that identifies the context uniquely. Although the context name is specified as a QName, it does not refer to an XML element. You can choose any unique QName as the context name.
4. The QName with namespace URI, `http://schemas.iona.com/types/context`, and local part, `SOAPHeaderInfo`, identifies the context type from Example 95 on page 180.
5. The QName with namespace URI, `http://schemas.iona.com/custom_header`, and local part, `header_content`, corresponds to the `message` attribute of a `soap:header` element. The value is currently ignored (but should not clash with any existing message QNames).
6. The `header_info` string value identifies the part of the SOAP header that holds the context data. It corresponds to the `part` attribute of a `<soap:header>` attribute. The value is currently ignored.
7. The call to `register_context()` tells the Artix Bus that the `SOAPHeaderInfo` type will be used to send context data in SOAP headers. After you have registered the context, the Bus is prepared to marshal the context data (if any) into a SOAP header.

Service Implementation

This subsection discusses the implementation of the `Greeter` port type, which maps to the `GreeterImpl` servant class in C++.

In the custom SOAP header demonstration, the `GreeterImpl::sayHi()` operation is modified to peek at the context data accompanying the invocation. To access the context data, you need to get access to a context current object, which encapsulates all of the context data received from the client.

Implementation of the sayHi operation

Example 100 shows the implementation of the `sayHi()` operation from the `GreeterImpl` servant class. The `sayHi()` operation implementation uses the context API to access the context data received from the client.

Example 100:*sayHi Operation Accessing a SOAP Context*

```
// C++
...
void
GreeterImpl::sayHi(
    IT_Bus::String &theResponse
) IT_THROW_DECL((IT_Bus::Exception))
{
    cout << "sayHi invoked" << endl;
    theResponse = "Hello from Artix";

    // Obtain a pointer to the bus
    Bus_var bus = Bus::create_reference();

1   ContextRegistry* context_registry =
        bus->get_context_registry();

    // Create QName objects needed to define a context
2   const QName principal_ctx_name(
        "",
        "SOAPHeaderInfo",
        ""
    );

    // Obtain a reference to the ContextCurrent
3   ContextCurrent& context_current =
        context_registry->get_current();

    // Obtain a pointer to the RequestContextContainer
4   ContextContainer* context_container =
        context_current.request_contexts();

    // Obtain a reference to the context
5   AnyType* info = context_container->get_context(
        principal_ctx_name
    );

    // Cast the context into a SOAPHeaderInfo object
6   SOAPHeaderInfo* header_info =
        dynamic_cast<SOAPHeaderInfo*> (info);

    // Extract the application specific SOAP header information
7   String& originator = header_info->getoriginator();
    String& message = header_info->getmessage();

   cout << "SOAP Header originator = " << originator.c_str() <<
  endl;
    cout << "SOAP Header message = " << message.c_str() << endl;
}
```

The preceding code example can be explained as follows:

1. The `IT_Bus::ContextRegistry` object, `context_registry`, provides access to all of the objects associated with contexts.
2. The QName with local name, `SOAPHeaderInfo`, is the name of the context to be extracted from the incoming request message.
3. Call `IT_Bus::ContextRegistry::get_current()` to obtain the `IT_Bus::ContextCurrent` object for the current thread.
4. Call `IT_Bus::ContextCurrent::request_contexts()` to obtain the `IT_Bus::ContextContainer` object containing all of the incoming request contexts.

   Note: This is the same object that is used on the client side to hold all of the outgoing request contexts.

5. To retrieve a specific context from the request context container, pass the context's name into the `IT_Bus::ContextContainer::get_context()` function.
6. The `IT_Bus::AnyType` class is the base type for all types in Artix. In this example, you can cast the `AnyType` instance, `info`, to its derived type, `SOAPHeaderInfo`.
7. You can now access the context data by calling the accessors for the `originator` and `message` elements, `getoriginator()` and `getmessage()`.

CORBA Header Contexts

This section describes how to propagate context data in a CORBA header, giving code examples for a consumer and a service provider.

Custom CORBA Header Scenario

Figure 27 shows an overview of the custom CORBA header scenario, showing how the client piggybacks context data along with an invocation request that is invoked on the sayHi operation.

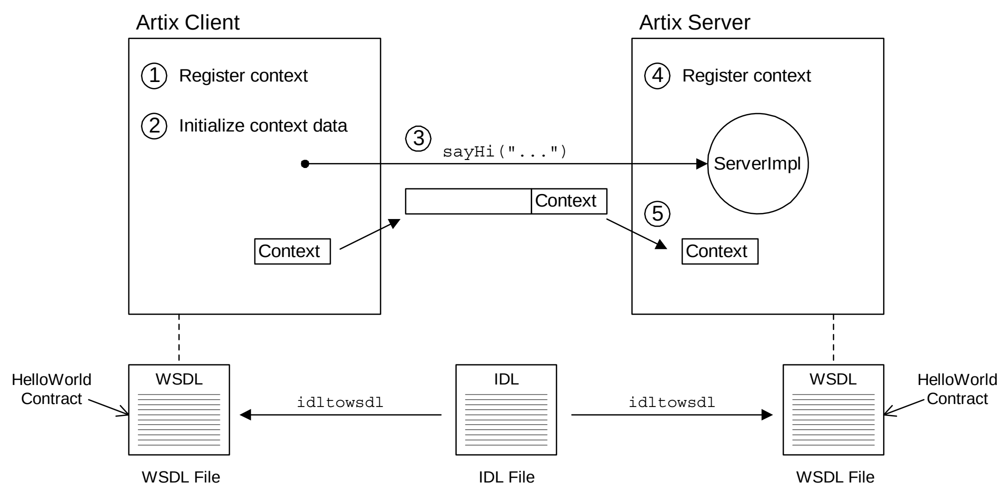

Figure 27: *Overview of the Custom CORBA Header Scenario*

Transmission of context data

As illustrated in Figure 27, CORBA context data is transmitted as follows:

1. The client registers the context type, PrincipalInfo, with the Bus.
2. The client initializes the context data instance.
3. The client invokes the sayHi() operation on the server.
4. As the server starts up, it registers the PrincipalInfo context type with the Bus.
5. When the sayHi() operation request arrives on the server side, the sayHi() operation implementation extracts the context data from the request.

HelloWorld IDL contract

Because this client-server application uses the CORBA binding, the HelloWorld IDL contract is originally written in OMG IDL, not WSDL. The following entities are defined in the IDL contract:

- HelloWorld *interface*—defines the interface to the service implemented on the server side (defining the IDL operations: sayHi and greetMe).
- PrincipalInfo *struct*—is used as the context data type. At runtime, an instance of PrincipalInfo type is transmitted in the CORBA header (in a GIOP service context). See Example 101 on page 194 for details.

HelloWorld WSDL contract

The HelloWorld WSDL contract is generated from the OMG IDL contract by invoking the Artix `idltowsdl` command-line tool.

Request and reply contexts

Artix supports the sending of context data both in request messages and in reply messages. The example scenario described here, however, only demonstrates how to send context data in CORBA requests.

CORBA Service Contexts

In the CORBA standard, the mechanism for sending header data is defined by the General Inter-ORB Protocol (GIOP). You can send custom header data in a GIOP header by encapsulating your data inside a *GIOP service context*. A GIOP service context consists of the following parts:

- *Service context ID*—a 32-bit integer ID that uniquely identifies the header type.
- *Service context data*—the custom data that you want to send. Formally, the service context data is an opaque block of binary data (preceded by a 32-bit integer, which gives the length of the block). In practice, however, it is usual to encode the data in this block using the Common Data Representation (CDR), which is part of the GIOP standard.

Selecting a service context ID

You must exercise care when selecting a service context ID, to ensure that it does not clash with the IDs defined by the OMG or other organizations. To avoid clashing IDs, the OMG allocates ID ranges in tranches of length 4096. The lowest range of IDs, 0–4095, is reserved for use by the OMG. To select a service context ID that is guaranteed not to clash with IDs used by other organizations, proceed as follows:

1. Apply to the OMG (www.omg.org), requesting them to allocate a tranche of 4096 service context IDs. The OMG will allocate you a 20-bit vendor service context codeset ID (VSCID), which defines the 20 high-order bits of the 32-bit service context ID.

   For example, has the VSCID, `0x49545xxx`.
2. The low-order 12 bits define the rest of the service context ID (giving a maximum of 4096 distinct IDs). You are responsible for allocating the low-order bits of the ID within your organization.

Defining service context data

Normally, you define a service context data type in the OMG IDL language. This is the logical approach to use, because service contexts are conventionally encoded using CDR, which maps OMG IDL data types to binary format.

For example, in the custom CORBA header scenario, the service context data type, `PrincipalInfo`, is defined in OMG IDL as follows:

Example 101:*PrincipalInfo Data Type Defined in OMG IDL*

```
// OMG IDL
struct PrincipalInfo
{
    string username;
    string password;
};
```

Where the OMG IDL `struct` type is analogous to an XML schema `sequence` type.

Converting the service context data type to WSDL

In order to manipulate the service context data from within an Artix program, it is necessary to convert the service context data type (which is defined in OMG IDL) to WSDL.

To perform the IDL-to-WSDL conversion, invoke the Artix `idltowsdl` command-line utility as follows:

```
idltowsdl HelloWorld.idl
```

Where the `HelloWorld.idl` file contains the definition of the `PrincipalInfo` struct type (along with definitions of other IDL data types and interfaces). After performing the conversion, the output file, `HelloWorld.wsdl`, contains the following definitions:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated by <idltowsdl> Tool. Version 4.2.0 -->
<definitions name="HeaderType"
    targetNamespace="http://schemas.iona.com/idl/HeaderType.idl"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:corba="http://schemas.iona.com/bindings/corba"
    xmlns:corbatm="http://schemas.iona.com/typemap/corba/HeaderType.idl"
    xmlns:tns="http://schemas.iona.com/idl/HeaderType.idl"
    xmlns:wsa="http://www.w3.org/2005/08/addressing"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://schemas.iona.com/idltypes/HeaderType.idl">
    <types>
       <schema targetNamespace="http://schemas.iona.com/idltypes/HeaderType.idl"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
            <complexType name="PrincipalInfo">
                <sequence>
                    <element name="username" type="string"/>
                    <element name="password" type="string"/>
```

```
                </sequence>
            </complexType>
        </schema>
    </types>
    <corba:typeMapping
        targetNamespace="http://schemas.iona.com/typemap/corba/HeaderType.idl">
        <corba:struct name="PrincipalInfo" repositoryID="IDL:PrincipalInfo:1.0"
            type="xsd1:PrincipalInfo">
            <corba:member idltype="corba:string" name="username"/>
            <corba:member idltype="corba:string" name="password"/>
        </corba:struct>
    </corba:typeMapping>
</definitions>
```

Configuration Prerequisites

To enable the propagation of context data in a CORBA header, it is a prerequisite to include the `CORBA_CONTEXT` interceptor in the `binding:client_binding_list` and `binding:server_binding_list` settings in your Artix configuration file.

Note: The `CORBA_CONTEXT` interceptor is an *ART interceptor* (a type of interceptor specific to the CORBA binding), *not* a regular Artix interceptor. The role of this interceptor is to move header data back and forth between the CORBA binding layer and the Artix service context layer.

Client binding list

Example 102 shows how to configure the *client binding list* to make GIOP headers accessible to Artix clients. You can apply this setting at the root scope of the Artix configuration file (for example, in `artix.cfg`).

Example 102: *Client Configuration Required for Using CORBA Headers*

```
# Artix Configuration File
...
binding:client_binding_list =
   ["OTS+CORBA_CONTEXT+TLS_Coloc+POA_Coloc",
   "CORBA_CONTEXT+TLS_Coloc+POA_Coloc",
   "OTS+CORBA_CONTEXT+POA_Coloc", "CORBA_CONTEXT+POA_Coloc",
   "CSI+OTS+CORBA_CONTEXT+GIOP+IIOP_TLS",
   "OTS+CORBA_CONTEXT+GIOP+IIOP_TLS",
   "CSI+CORBA_CONTEXT+GIOP+IIOP_TLS",
   "CORBA_CONTEXT+GIOP+IIOP_TLS",
   "CSI+OTS+CORBA_CONTEXT+GIOP+IIOP",
   "OTS+CORBA_CONTEXT+GIOP+IIOP", "CSI+CORBA_CONTEXT+GIOP+IIOP",
   "CORBA_CONTEXT+GIOP+IIOP"];
```

Server binding list

Example 103 shows how to configure the *server binding list* to GIOP headers accessible to Artix servers.

Example 103:*Server Configuration Required for Using CORBA Headers*

```
# Artix Configuration File
...
binding:server_binding_list = ["OTS+CORBA_CONTEXT", "OTS",""];
```

Client Main Function

This subsection discusses the client for the custom CORBA header scenario. This client is designed to send a custom header, of `PrincipalInfo` type, every time it invokes an operation on the `HelloWorld` port type.

To enable the sending of context data, the client performs two fundamental tasks, as follows:

1. *Register a context type with the context registry*—registering the context type is a prerequisite for sending context data in a request. By registering the context type with the Bus, you give the Bus instance the capability to marshal and unmarshal context data of that type.
2. *Initialize the context data in the ContextCurrent object*—before invoking any operations, the client obtains an instance of the header context data from an `IT_Bus::ContextCurrent` object. After initializing the header context data, any operations invoked from the current thread will include the header context data.

Client main function

Example 104 shows sample code from the client main function, which shows how to register a context type and initialize header context data for the current thread.

Example 104:*Client Main Function Setting a CORBA Context*

```
// C++
// HelloWorldClientSample.cxx File

#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_cal/iostream.h>

// Include header files related to Artix contexts
1 #include <it_bus_pdk/context.h>

// Include header files representing the CORBA header content
2 #include "HelloWorld_wsdlTypes.h"
#include "HelloWorld_wsdlTypesFactory.h"
#include "HelloWorldClient.h"
```

Example 104: *Client Main Function Setting a CORBA Context*

```
IT_USING_NAMESPACE_STD

using namespace IT_Bus;
using namespace IT_WS_ORB;
using namespace IT_ContextAttributes;

int
main(int argc, char* argv[])
{
    try
    {
        IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);
        HelloWorldClient client;

3       ContextRegistry* context_registry =
            bus->get_context_registry();

        // Create QName objects needed to define a context
4       const QName ctx_name(
            "",
            "PrincipalInfo",
            ""
        );
5       const QName ctx_type(
            "",
            "PrincipalInfo",
   "http://schemas.iona.com/idltypes/HelloWorld.idl"
        );
6       const unsigned long ctx_id = 12288;

        // Register the context with the ContextRegistry
7       context_registry->register_context(
            ctx_name,
            ctx_type,
            ctx_id
        );

        // Obtain a reference to the ContextCurrent
8       ContextCurrent& context_current =
            context_registry->get_current();

        // Obtain a pointer to the RequestContextContainer
9       ContextContainer* context_container =
            context_current.request_contexts();

        // Obtain a reference to the context
10      AnyType* info = context_container->get_context(
            ctx_name,
            true
        );

        // Cast the context into a PrincipalInfo object
11      PrincipalInfo* header_info =
            dynamic_cast<PrincipalInfo*> (info);

        // Add the header content
        header_info->setusername("Bill");
        header_info->setpassword("Rendezvous");
```

Example 104: *Client Main Function Setting a CORBA Context*

```
        // Invoke the Web service business methods
        String theResponse;

12      client.sayHi(theResponse);
        cout << "sayHi response: " << theResponse << endl;
    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Error : Unexpected error occured!"
             << endl << e.message()
             << endl;
        return -1;
    }
    return 0;
}
```

The preceding code example can be explained as follows:

1. The `it_bus_pdk/context.h` header file contains the declarations of the following classes:
   - `IT_Bus::ContextRegistry`,
   - `IT_Bus::ContextContainer`,
   - `IT_Bus::ContextCurrent`.
2. The `HelloWorld_wsdlTypes.h` local header file contains the declaration of the `PrincipalInfo` class, which has been generated from the context schema (see Example 95 on page 180).
3. Obtain a reference to the `IT_Bus::ContextRegistry` object, which is used to register contexts with the Bus.
4. The QName with local name, `PrincipalInfo`, is a context name that identifies the context uniquely. Although the context name is specified as a QName, it does not refer to an XML element. You can choose any unique QName as the context name.
5. The QName with namespace URI, `http://schemas.iona.com/idltypes/HelloWorld.idl`, and local part, `PrincipalInfo`, identifies the context type from Example 95 on page 180.
6. The `ctx_id` specifies the ID of the GIOP service context that will hold the context data. For more details about GIOP service contexts, see “CORBA Service Contexts” on page 193.
7. The call to `register_context()` tells the Artix Bus that the `PrincipalInfo` type will be used to send context data in a GIOP service context. After you have registered the context, the Bus is prepared to marshal the context data (if any) into a CORBA header.
8. Call `IT_Bus::ContextRegistry::get_current()` to obtain a reference to the `IT_Bus::ContextCurrent` object. The current object provides access to all context objects associated with the current thread.
9. Call `IT_Bus::ContextContainer::request_contexts()` to obtain an `IT_Bus::ContextContainer` object that contains all of the contexts for requests originating from the current thread.

10. The `IT_Bus::ContextContainer::get_context()` function is called with its second parameter set to `true`, indicating that a context with that name should be created if none already exists.
11. The `IT_Bus::AnyType` class is the base type for all complex types in Artix. In this case, you can cast the `AnyType` instance, `info`, to its derived type, `PrincipalInfo*`.

    By setting the `username` and `password` elements of this `PrincipalInfo` object, you are effectively fixing the context data for all operations invoked from this thread.
12. When you invoke the `sayHi()` operation, the context data is included in the CORBA header. From this point on, any WSDL operation invoked from the current thread will include the `PrincipalInfo` context data in its CORBA header.

Server Main Function

This subsection discusses the main function for the server in the custom CORBA header scenario. In addition to the usual boilerplate code for an Artix server (that is, registering a servant and calling `IT_Bus::run()`), this server also registers a context type with the Bus.

By registering a context type with the Bus, you give the Bus instance the capability to unmarshal context data of that type. This unmarshalling capability is then exploited in the implementation of the `sayHi()` operation (see Example 106 on page 201).

Server main function

Example 105 shows sample code from the server main function, which registers the `PrincipalInfo` context type and then creates and registers a `HelloWorldImpl` servant object.

Example 105: *Server Main Function Registering a CORBA Context*

```
// C++
#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_bus/fault_exception.h>
#include <it_cal/iostream.h>

1 #include <it_bus_pdk/context.h>

#include "HelloWorldImpl.h"

IT_USING_NAMESPACE_STD

using namespace IT_Bus;

int
main(int argc, char* argv[])
{
    try
    {
        IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);
```

Example 105: *Server Main Function Registering a CORBA Context*

```
2           ContextRegistry* context_registry =
                bus->get_context_registry();

3           const QName ctx_name(
                "",
                "PrincipalInfo",
                ""
            );
4           const QName ctx_type(
                "",
                "PrincipalInfo",
            "http://schemas.iona.com/idltypes/HelloWorld.idl"
            );
5           const unsigned long ctx_id = 12288;

6           context_registry->register_context(
                ctx_name,
                ctx_type,
                ctx_id
            );

            HelloWorldImpl servant(bus);

          IT_Bus::QName service_name("", "HelloWorldCORBAService",
      "http://schemas.iona.com/idl/HelloWorld.idl");

            bus->register_servant(
                servant,
                "../../etc/hello_world.wsdl",
                service_name
            );

            IT_Bus::run();
        }
        catch(IT_Bus::Exception& e)
        {
            cout << "Error occurred: " << e.message() << endl;
            return -1;
        }
        return 0;
}
```

The preceding code example can be explained as follows:

1. The `it_bus_pdk/context.h` header file contains the declarations of the following classes:
   - `IT_Bus::ContextRegistry`,
   - `IT_Bus::ContextContainer`,
   - `IT_Bus::ContextCurrent`.
2. Obtain a reference to the `IT_Bus::ContextRegistry` object, which is used to register contexts with the Bus.
3. The QName with local name, `PrincipalInfo`, is a context name that identifies the context uniquely. Although the context name is specified as a QName, it does not refer to an XML element. You can choose any unique QName as the context name.

4. The QName with namespace URI, `http://schemas.iona.com/idltypes/HelloWorld.idl`, and local part, `PrincipalInfo`, identifies the context type from Example 101 on page 194.
5. The `ctx_id` specifies the ID of the GIOP service context that holds the context data. For more details about GIOP service contexts, see "CORBA Service Contexts" on page 193.
6. The call to `register_context()` tells the Artix Bus that the `PrincipalInfo` type will be used to send context data in CORBA headers. After you have registered the context, the Bus is prepared to marshal the context data (if any) into a CORBA header.

Service Implementation

This subsection discusses the implementation of the `HelloWorld` port type, which maps to the `HelloWorldImpl` servant class in C++.

In the custom CORBA header scenario, the `HelloWorldImpl::sayHi()` operation is modified to peek at the context data accompanying the invocation. To access the context data, you need to get access to a context current object, which encapsulates all of the context data received from the client.

Implementation of the sayHi operation

Example 106 shows the implementation of the `sayHi()` operation from the `HelloWorldImpl` servant class. The `sayHi()` operation implementation uses the context API to access the context data received from the client.

Example 106: *sayHi Operation Accessing a CORBA Context*

```
// C++
...
void
GreeterImpl::sayHi(
    IT_Bus::String &theResponse
) IT_THROW_DECL((IT_Bus::Exception))
{
    cout << "sayHi invoked" << endl;
    theResponse = "Hello from Artix";

    // Obtain a pointer to the bus
    Bus_var bus = Bus::create_reference();

1   ContextRegistry* context_registry =
        bus->get_context_registry();

    // Create QName objects needed to define a context
2   const QName ctx_name(
        "",
        "PrincipalInfo",
        ""
    );
```

Example 106: *sayHi Operation Accessing a CORBA Context*

```
    // Obtain a reference to the ContextCurrent
3   ContextCurrent& context_current =
        context_registry->get_current();

    // Obtain a pointer to the Request ContextContainer
4   ContextContainer* context_container =
        context_current.request_contexts();

    // Obtain a reference to the context
5   AnyType* info = context_container->get_context(
        ctx_name
    );

    // Cast the context into a PrincipalInfo object
6   PrincipalInfo* header_info =
        dynamic_cast<PrincipalInfo*> (info);

    // Extract the application specific CORBA header information
7   String& username = header_info->getusername();
    String& password = header_info->getpassword();

    cout << "CORBA Header username = "
         << originator.c_str() << endl;
    cout << "CORBA Header password = "
         << message.c_str() << endl;
}
```

The preceding code example can be explained as follows:

1. The `IT_Bus::ContextRegistry` object, `context_registry`, provides access to all of the objects associated with contexts.
2. The QName with local name, `PrincipalInfo`, is the name of the context to be extracted from the incoming request message.
3. Call `IT_Bus::ContextRegistry::get_current()` to obtain the `IT_Bus::ContextCurrent` object for the current thread.
4. Call `IT_Bus::ContextCurrent::request_contexts()` to obtain the `IT_Bus::ContextContainer` object containing all of the incoming request contexts.

   Note: This is the same object that is used on the client side to hold all of the outgoing request contexts.

5. To retrieve a specific context from the request context container, pass the context's name into the `IT_Bus::ContextContainer::get_context()` function.
6. The `IT_Bus::AnyType` class is the base type for all types in Artix. In this example, you can cast the `AnyType` instance, `info`, to its derived type, `PrincipalInfo*`.
7. You can now access the context data by calling the accessors for the `username` and `password` elements, `getusername()` and `getpassword()`.

Header Contexts in Three-Tier Systems

This section considers how Artix header contexts are propagated in a three-tier system. The Artix context model makes no distinction between *incoming* request contexts and *outgoing* request contexts. Similarly, Artix makes no distinction between *incoming* reply contexts and *outgoing* reply contexts. An implicit consequence of this model is that request contexts and reply contexts are automatically propagated across multiple application tiers.

Request context propagation

Figure 28 shows an example of a three-tier system where a request context is propagated automatically from tier to tier.

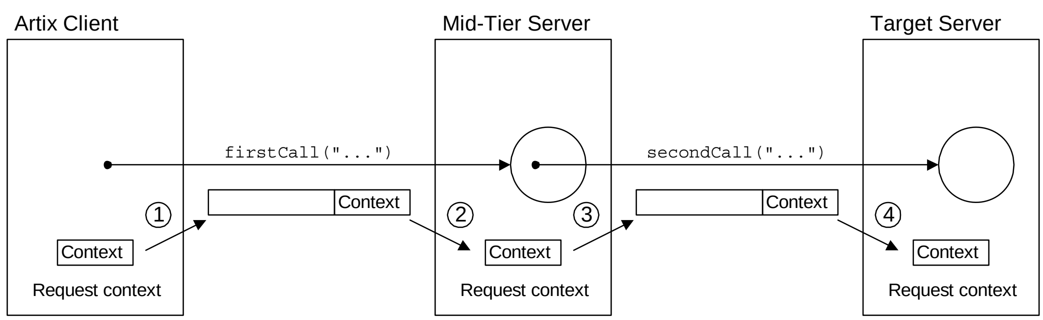

Figure 28: *Propagation of a Request Context in a Three-Tier System*

Context propagation steps

In Figure 28, the request context is propagated through the three-tier system as follows:

1. In the Artix client, a header context is added to the request context container. When the client makes an invocation, `firstCall()`, on the mid-tier, the context is inserted into the request message header.
2. When the request arrives at the mid-tier, it is automatically marshalled into a request context. The context data is now accessible using the request context container object.
3. If the mid-tier makes a follow-on invocation, `secondCall()`, the Artix runtime inserts the received request context into the outgoing request message. Hence, the client's request context is automatically forwarded on to the next tier.
4. When the request arrives at the target, it is automatically marshalled into a request context. The client context data is now accessible through the request context container object.

Working with Transport Attributes

Using the Artix context mechanism, you can set many of the the transport attributes at runtime.

How Artix Stores Transport Attributes

Artix uses the context mechanism described in "Artix Contexts" on page 153 to store the properties used to configure the transport layer and populate any headers used by the selected transport. Most of the properties are stored in the request and reply context containers. However, some properties that are used in initializing the transport layer at start-up are stored in a special context container, the *configuration context container*.

Initialization properties

Some transport attributes, such as JMS broker sign-on values or a server's HTTP endpoint URL, are used by Artix when it is initializing the transport layer. Therefore, they need to be specified *before* Artix initializes the transport layer for a service or a service proxy. These attributes are stored in a configuration context container. When the bus initializes the transport layer, it will check the configuration context container for any initialization properties.

Global transport attributes

For most transport properties such as HTTP keep-alive, WebSphere MQ `AccessMode`, and Tib/RV `callbackLevel`, the context objects containing the transport's properties are stored in the Artix request context container and the Artix reply context container. Once you have retrieved the context object from the proper context container, you can inspect the values of transport headers and other transport related properties such as codeset conversion. You can also dynamically set many of the values for outgoing messages using the context APIs. For a full listing of all the possible port attributes for each transport see the *Artix WSDL Reference*.

Transport specific

Transport attributes are stored in built-in contexts. These contexts are preregistered with the context container when the transport layer is initialized. They are specific to the different transports. For example, if you request the context for the HTTP port attributes from the context container, the returned context will have methods for setting and examining HTTP specific attributes. However, if the application is using another transport, WebSphere

MQ for example, the HTTP configuration context will not be registered and you will be unable to get the HTTP configuration context from the container.

Default values

All of the transport attributes have default values that are specified in either the service's contract or in the service's configuration. If you do not use the contexts for overriding transport attributes, these defaults are used when sending messages.

When are the attribute contexts populated

Whether or not an attribute context is populated when you access it depends on whether the context was taken from an outgoing message or an incoming message, as follows:

- *Outgoing messages*—when you get the transport attributes for an outgoing message, the context is empty. You need to create an instance of the context and set the values you want to override in the context yourself.
- *Incoming messages*—when a message is received by the transport layer, the transport populates the context with the attributes of the message it receives.

  For example, if you are using HTTP, the values of the incoming message's HTTP header are used to populate the context. The context can then be inspected at any point in the application's code.

Getting and Setting Transport Attributes

The contexts for holding transport attributes are handled using either the standard context mechanism or the configuration context mechanism. To get a transport attribute context do the following:

1. Make sure you include the requisite header file for the transport attribute context.
2. Use the context API to obtain either a request context container, a reply context container, or a configuration context container, as appropriate.
3. Call `get_context()` on the context container, passing in the QName of the transport attribute context.
4. Cast the returned context data to the appropriate type.

Once you have the context data you can inspect it and set new values for any of its properties.

Schemas directory

The schemas for the Artix configuration contexts are located in the following directory:

ArtixInstallDir/schemas

Header files

The header files for the Artix configuration contexts are located in the following directory:

ArtixInstallDir/include/it_bus_pdk/context_attrs

Library

To gain access to the context stubs, you should link with the following library:

Windows

ArtixInstallDir/lib/it_context_attribute.lib

UNIX

ArtixInstallDir/lib/it_context_attribute.so
ArtixInstallDir/lib/it_context_attribute.sl

Headers and types for the pre-registered contexts

The following list gives the context name, data type and header file for each of the pre-registered contexts. The name of each context is a C++ constant of `IT_Bus::QName` type, defined in the `IT_ContextAttributes` namespace (for example, `IT_ContextAttributes::HTTP_CLIENT_OUTGOING_CONTEXTS`). You can pass the context name as a parameter to the `IT_Bus::ContextContainer::get_context()` function to obtain a pointer to the context data.

HTTP client outgoing attributes

This context enables you to specify HTTP context data for inclusion with the next outgoing client request. Table 2 shows the relevant details for accessing this context.

Table 2: *Details for HTTP Client Outgoing Context*

Description	Value
Header file	`<it_bus_pdk/context_attrs/http_conf_xsdTypes.h>`
Kind of context container	Request
Context QName	`IT_ContextAttributes::HTTP_CLIENT_OUTGOING_CONTEXTS`
Type of context data	`IT_ContextAttributes::clientType`

HTTP client incoming attributes

This context enables you to read context data received with the last HTTP reply on the client side. Table 3 shows the relevant details for accessing this context.

Table 3: *Details for HTTP Client Incoming Context*

Description	Value
Header file	`<it_bus_pdk/context_attrs/http_conf_xsdTypes.h>`
Kind of context container	Reply
Context QName	`IT_ContextAttributes::HTTP_CLIENT_INCOMING_CONTEXTS`
Type of context data	`IT_ContextAttributes::clientType`

HTTP server outgoing attributes

This context enables you to specify HTTP context data for inclusion with the server's reply. Table 4 shows the relevant details for accessing this context.

Table 4: *Details for HTTP Server Outgoing Context*

Description	Value
Header file	`<it_bus_pdk/context_attrs/http_conf_xsdTypes.h>`
Kind of context container	Reply
Context QName	`IT_ContextAttributes::HTTP_SERVER_OUTGOING_CONTEXTS`
Type of context data	`IT_ContextAttributes::serverType`

HTTP server incoming attributes

This context enables you to read context data received with the current HTTP request on the server side. Table 5 shows the relevant details for accessing this context.

Table 5: *Details for HTTP Server Incoming Context*

Description	Value
Header file	`<it_bus_pdk/context_attrs/http_conf_xsdTypes.h>`
Kind of context container	Request
Context QName	`IT_ContextAttributes::HTTP_SERVER_INCOMING_CONTEXTS`
Type of context data	`IT_ContextAttributes::serverType`

CORBA transport attributes

This context can be used to access and modify the CORBA Principal. Table 6 shows the relevant details for accessing this context.

Table 6: *Details for CORBA Transport Context*

Description	Value
Header file	<it_bus_pdk/context_attrs/corba_xsdTypes.h>
Kind of context container	Request, Reply
Context QName	`IT_ContextAttributes::CORBA_CONTEXT_ATTRIBUTES`
Type of context data	`IT_ContextAttributes::CORBAAttributesType`

Principal attribute

Calling `get_context()` returns the Principal as an `IT_Bus::StringHolder` instance. Table 7 shows the relevant details for accessing this context.

Table 7: *Details for Principal Context*

Description	Value
Header file	<it_bus_pdk/context_attrs/context_types.h>
Kind of context container	Request, Reply
Context QName	`IT_ContextAttributes::PRINCIPAL_CONTEXT_ATTRIBUTE`
Type of context data	`IT_Bus::StringHolder`

MQ connection attributes

This context is used to set MQ connection attributes on the client side of a connection. After each invocation, the connection attributes are changed back to the defaults specified in the WSDL contract. Table 8 shows the relevant details for accessing this context.

Table 8: *Details for MQ Connection Attributes Context*

Description	Value
Header file	<it_bus_pdk/context_attrs/mq_xsdTypes.h>
Kind of context container	Request
Context QName	`IT_ContextAttributes::MQ_CONNECTION_ATTRIBUTES`
Type of context data	`IT_ContextAttributes::MQConnectionAttributesType`

MQ outgoing message attributes

For a client, this context enables you to set the MQ message attributes on the next outgoing request. For a server, this context enables you to set the MQ message attributes on the next outgoing reply. Table 9 shows the relevant details for accessing this context.

Table 9: *Details for MQ Outgoing Message Attributes Context*

Description	Value
Header file	`<it_bus_pdk/context_attrs/mq_xsdTypes.h>`
Kind of context container	Request, Reply
Context QName	`IT_ContextAttributes::MQ_OUTGOING_MESSAGE_ATTRIBUTES`
Type of context data	`IT_ContextAttributes::MQMessageAttributesType`

MQ incoming message attributes

For a client, this context enables you to read the MQ message attributes received from the last reply. For a server, this context enables you to read the MQ message received with the current request. Table 10 shows the relevant details for accessing this context.

Table 10: *Details for MQ Incoming Message Attributes Context*

Description	Value
Header file	`<it_bus_pdk/context_attrs/mq_xsdTypes.h>`
Kind of context container	Request, Reply
Context QName	`IT_ContextAttributes::MQ_INCOMING_MESSAGE_ATTRIBUTES`
Type of context data	`IT_ContextAttributes::MQMessageAttributesType`

FTP connection policy

For clients and servers, you can set all of the FTP connection policies in a configuration context. For a client, you can additionally set the scan interval policy and the receive timeout policy in a request context. Table 11 shows the relevant details for accessing this context.

Table 11: *Details for FTP Connection Policy Context*

Description	Value
Header file	<it_bus_pdk/context_attrs/ftp_context_xsdTypes.h>
Kind of context container	Configuration, Request
Context QName	`IT_ContextAttributes::FTP_CONNECTION_POLICY`

Table 11: *Details for FTP Connection Policy Context*

Description	Value
Type of context data	`IT_ContextAttributes::ConnectionPolicyType`

FTP connection credentials

For clients and servers, the FTP connection credentials context enables you to set username and password for opening a connection to the FTP daemon. Table 12 shows the relevant details for accessing this context.

Table 12: *Details for FTP Connection Credentials Context*

Description	Value
Header file	<it_bus_pdk/context_attrs/ftp_context_xsdTypes.h>
Kind of context container	Configuration
Context QName	`IT_ContextAttributes::FTP_CREDENTIALS`
Type of context data	`IT_ContextAttributes::CredentialsType`

FTP client naming policy

The FTP client naming policy enables you to register a class that generates the names of the files created to store messages in the FTP file system. Because this class must be a Java class, it is only possible to use this feature from an Artix Java application. See *Developing Artix Applications in Java* for details.

FTP server naming policy

The FTP server naming policy enables you to register a class that generates the names of the files created to store messages in the FTP file system. Because this class must be a Java class, it is only possible to use this feature from an Artix Java application. See *Developing Artix Applications in Java* for details.

i18n server attributes

For a server, the i18n server attributes context enables you to set the local codeset and the server outbound codeset in the reply context. Table 13 shows the relevant details for accessing this context.

Table 13: *Details for I18N Server Attributes Context*

Description	Value
Header file	`<it_bus_pdk/context_attrs/i18n_context_xsdTypes.h>`
Kind of context container	Reply
Context QName	`IT_ContextAttributes::I18N_INTERCEPTOR_SERVER_QNAME`
Type of context data	`IT_ContextAttributes::ServerConfiguration`

i18n client attributes

For a server, the i18n client attributes context enables you to set the local codeset and the client outbound codeset in the request context. Table 14 shows the relevant details for accessing this context.

Table 14: *Details for I18N Client Attributes Context*

Description	Value
Header file	`<it_bus_pdk/context_attrs/i18n_context_xsdTypes.h>`
Kind of context container	Request
Context QName	`IT_ContextAttributes::I18N_INTERCEPTOR_CLIENT_QNAME`
Type of context data	`IT_ContextAttributes::ClientConfiguration`

Bus security attributes

For clients and servers, enables you to set security attributes programmatically. Table 15 shows the relevant details for accessing this context.

Table 15: *Details for Bus Security Attributes Context*

Description	Value
Header file	`<it_bus_pdk/context_attrs/bus_security_xsdTypes.h>`
Kind of context container	Request, Reply
Context QName	`IT_ContextAttributes::SECURITY_SERVER_CONTEXT`
Type of context data	`IT_ContextAttributes::BusSecurity`

HTTP endpoint URL attribute

For clients, this attribute enables you to specify the URL that will be used by the next proxy to open a HTTP connection. The context value is cleared after the proxy connection is opened. Table 16 shows the relevant details for accessing this context.

Table 16: *Details for HTTP Endpoint URL Context*

Description	Value
Header file	`<it_bus_pdk/context_attrs/context_types.h>`
Kind of context container	Request
Context QName	`IT_ContextAttributes::HTTP_ENDPOINT_URL`
Type of context data	`IT_Bus::StringHolder`

Server address context attributes

For servers, this context is set only when you have registered a default servant (see "Default Servants" on page 677). By reading this context from the request context container, the server can determine the identity of the target service. Table 17 shows the relevant details for accessing this context.

Table 17: *Details for Server Address Context*

Description	Value
Header file	`<it_bus_pdk/context_attrs/address_context.h>`
Kind of context container	Request
Context QName	`IT_ContextAttributes::SERVER_ADDRESS_CONTEXT`
Type of context data	`IT_ContextAttributes::AddressContext`

Server operation attribute

This context is a non-serializable context that can be used to get a reference to an `IT_Bus::ServerOperation` object during an invocation on the server side. In other words, you can access this context type from the body of a servant function. See "Reading and Writing Custom Types" on page 168 for more details about non-serializable contexts.

Table 18: *Details for Server Operation Context*

Description	Value
Header file	`<it_bus_pdk/context_attrs/context_types.h>`
Kind of context container	Request
Context QName	`IT_ContextAttributes::SERVER_OPERATION_CONTEXT`

Table 18: *Details for Server Operation Context*

Description	Value
Type of context data	`IT_Bus::ServerOperationContext`

Getting IP Attributes

Artix provides a context that enables you to access data from the IP socket layer. Currently, the only supported IP attribute is the client IP address, which is accessible through the *client address context*.

Client address context

The client address context is a server-side request context that contains the IP address (or hostname) of the requesting client. This context can be useful if you want a simple way of identifying clients—for example, for the purposes of logging requests on the server side.

> **WARNING:** The client address context is *not* a secure way to identify clients. If you need to be certain of the client's identity, use one of the authentication techniques described in the ***Artix Security Guide, C++***.

Enabling the client address context

To enable the client address context on the server side, insert the following setting into the relevant scope of your server's `.cfg` configuration file:

```
# Artix Configuration File
plugins:bus:register_client_context = "true";
```

This setting causes the Bus to read the client's IP address from the IP socket layer each time the server receives a message from a client. The IP address is then inserted into a client address context, which is accessible to the server application code.

> **Note:** The default setting is `false`, thus disabling the client address context. This is to avoid any unnecessary performance overhead when this feature is not needed.

Getting the client address on the server side

The context containing the client's IP address, `CLIENT_ADDRESS_CONTEXT`, is available in the server's request context container, *after* a request from the client is received by the transport layer. To access the client's IP address on the server side, use the code fragment shown in Example 107.

Example 107:*Reading the Client IP Address on the Server Side*

```
// C++
1 #include <it_bus_pdk/context.h>
#include <it_bus_pdk/context_attrs/context_constants.h>
...
IT_USING_NAMESPACE_STD

using namespace IT_ContextAttributes;
using namespace IT_Bus;
...
2 ContextRegistry* context_registry =
    bus->get_context_registry();

ContextCurrent& context_current =
    context_registry->get_current();

// Obtain a pointer to the Request ContextContainer
ContextContainer* context_container =
    context_current.request_contexts();

// Obtain a reference to the context
3 AnyType* info = context_container->get_context(
    IT_ContextAttributes::CLIENT_ADDRESS_CONTEXT,
    false
);

4 IT_Bus::StringHolder * str_holder =
   dynamic_cast<StringHolder *>(info);
IT_Bus::String * client_ip_address;
if(0 != str_holder)
{
    client_ip_address = &(str_holder->get());
}
```

The preceding code can be explained as follows:

1. Include header file for the general context classes and for the context constants.
2. Obtain a reference to a context container, `context_container`, that contains the server's request contexts.
3. Extract the client address context (identified by the constant, `CLIENT_ADDRESS_CONTEXT`) from the list of server request contexts.
4. Cast the returned context object to `IT_Bus::StringHolder` type and extract the client's IP address from the string holder.

Setting HTTP Attributes

Artix uses four contexts to support the HTTP transport. Two contexts support the server-side HTTP information. The server-side contexts are of `IT_ContextAttributes::serverType` type. The other two contexts support the client-side HTTP information. The client-side contexts are of `IT_ContextAttributes::clientType` type.

The information stored in the HTTP transport attribute contexts correlates to the values passed in an HTTP header.

Client-side Configuration

HTTP clients have access to both the values being passed in the HTTP header of the outgoing request and the values received in the HTTP header of the response. The information for each header is stored in a separate context.

Outgoing header information

On the client-side, the outgoing context, `HTTP_CLIENT_OUTGOING_CONTEXTS`, is available in the client's request context. Any changes made to values in the outgoing context are placed in the request's HTTP header and propagated to the server. For example, if you want to allow requests to be automatically redirected you could set the `AutoRedirect` attribute to `true` in the client's outgoing context. Example 108 shows the code for setting the `AutoRedirect` property for a client.

Example 108: *Setting a Client's AutoRedirect Property*

```
   // C++
1  #include <it_bus_pdk/context.h>
   #include <it_bus_pdk/context_attrs/http_conf_xsdTypes.h>
   ...
   IT_USING_NAMESPACE_STD

   using namespace IT_ContextAttributes;
   using namespace IT_Bus;
   ...
2  ContextRegistry* context_registry =
       bus->get_context_registry();

   ContextCurrent& context_current =
       context_registry->get_current();

   // Obtain a pointer to the Request ContextContainer
   ContextContainer* context_container =
       context_current.request_contexts();

3  // Obtain a reference to the context
   AnyType* info = context_container->get_context(
       IT_ContextAttributes::HTTP_CLIENT_OUTGOING_CONTEXTS, true
   );
```

Example 108: *Setting a Client's AutoRedirect Property*

```
// Cast the context into a clientType object
clientType* http_client_config =
    dynamic_cast<clientType*> (info);

4 http_client_config->setAutoRedirect(true);

// make proxy invocations
...
```

The code in Example 108 does the following:

1. Includes the header files for the general context classes and for the HTTP client context type.
2. Gets the client's context registry.
3. Gets the client's outgoing HTTP context from the request context container.
4. Sets the value of the `AutoRedirect` property to `true`.

Outgoing client attributes

Table 19 shows the attributes that are valid in the outgoing HTTP client context.

Table 19: *Outgoing HTTP Client Attributes*

HTTP Attribute	Artix APIs	Description

`Accept`	`String* getAccept()`
	`const String* getAccept() const`
	
	`void setAccept(const String* val)`
	`void setAccept(const String& val)`
		Specifies the MIME types the client can handle in a response.

`Accept-Encoding`	`String* getAcceptEncoding()`
	`const String* getAcceptEncoding() const`
	
	`void setAcceptEncoding( const String* val)`
	`void setAcceptEncoding( const String& val)`
		Specifies the types of content encoding the client can handle in a response. This property typically refers to compression mechanisms.

`Accept-Language`	`String* getAcceptLanguage()`
	`const String* getAcceptLanguage() const`
	
	`void setAcceptLanguage( const String* val)`
	`void setAcceptLanguage( const String& val)`
		Specifies the language the client prefers. Valid language tags combine an ISO language code and an ISO country code separated by a hyphen. For example, `en-US`.

Table 19: *Outgoing HTTP Client Attributes*

HTTP Attribute	Artix APIs	Description

`Authorization`	`String* getAuthorization()`
	`const String* getAuthorization() const`
	
	`void setAuthorization( const String* val)`
	`void setAuthorization( const String& val)`
		Specifies the credentials that will be used by the server to authorize requests from the client.

`AuthorizationType`	`String* getAuthorizationType()`
	`const String* getAuthorizationType() const`
	
	`void setAuthorizationType( const String* val)`
	`void setAuthorizationType( const String& val)`
		Specifies the name of the authentication scheme in use.

`AutoRedirect`	`Boolean* getAutoRedirect()`
	`const Boolean* getAutoRedirect() const`
	
	`void setAutoRedirect( const Boolean* val)`
	`void setAutoRedirect( const Boolean& val)`
		Specifies whether a request should be automatically redirected by the server. The default is `false` to specify that requests are not to be automatically redirected.

`BrowserType`	`String* getBrowserType()`
	`const String* getBrowserType() const`
	
	`void setBrowserType( const String* val)`
	`void setBrowserType( const String& val)`
		Specifies information about the browser from which the request originates. This property is also know as the user-agent.

Table 19: *Outgoing HTTP Client Attributes*

HTTP Attribute	Artix APIs	Description

`Cache-Control`	`String* getCacheControl()`
	`const String* getCacheControl() const`
	
	`void setCacheControl(`
	`const String* val)`
	`void setCacheControl(`
	`const String& val)`
		Specifies directives to caches along the request/response path.
		Valid values are:
		`no-cache`: caches must revalidate responses with the server. If response header fields are given, the restriction applies only to those header fields.
		`no-store`: caches must not store any part of a request or its response.
		`max-age`: the max age, in seconds, of an acceptible response.
		`max-stale`: the client will accept expired messages. If a value is given, it specifies the how many seconds after a response expires that the it is still acceptable. If no value is given, all stale responses are acceptable.
		`min-fresh`: the response must stay fresh for the given number of seconds.
		`no-transform`: caches must not modify the media type or the content location of a response.
		`only-if-cached`: caches should return only cached responses.

`ClientCertificate`	`String* getClientCertificate()`
	`const String* getClientCertificate() const`
	
	`void setClientCertificate(`
	`const String* val)`
	`void setClientCertificate(`
	`const String& val)`
		Specifies the full path to the PKCS12-encoded X509 certificate issued by the certificate authority for the client.

`ClientCertificateChain`	`String* getClientCertificateChain()`
	`const String* getClientCertificateChain() const`
	
	`void setClientCertificateChain(`
	`const String* val)`
	`void setClientCertificateChain(`
	`const String& val)`
		Specifies the full path to the file containing all of the certificates in the chain.

Table 19: *Outgoing HTTP Client Attributes*

HTTP Attribute	Artix APIs	Description

ClientPrivateKey	`String* getClientPrivateKey()`
	`const String* getClientPrivateKey() const`
	
	`void setClientPrivateKey( const String* val)`
	`void setClientPrivateKey( const String& val)`
		Specifies the full path to the PKCS12-encoded private key that corresponds to the X509 certificate specified by `ClientCertificate`.

ClientPrivateKeyPassword	`String* getClientPrivateKeyPassword()`
	`const String* getClientPrivateKeyPassword() const`
	
	`void setClientPrivateKeyPassword( const String* val)`
	`void setClientPrivateKeyPassword( const String& val)`
		Specifies the password used to decrypt the PKCS12-encoded private key.

Connection	`String* getConnection()`
	`const String* getConnection() const`
	
	`void setConnection( const String* val)`
	`void setConnection( const String& val)`
		Specifies whether a connection is to be kept open after each request/response transaction.
		Valid values are:
		`close`: the connection is closed after each transaction.
		`Keep-Alive`: the client would like the conneciton to remain open. Servers do not have to honor this request.

Cookie	`String* getCookie()`
	`const String* getCookie() const`
	
	`void setCookie(const String* val)`
	`void setCookie(const String& val)`
		Specifies a static cookie that is sent along with a request.
		Note: According to the HTTP 1.1 specification, HTTP cookies must contain US-ASCII characters.

Expires	`String* getExpires()`
	`const String* getExpires() const`
	
	`void setExpires(const String* val)`
	`void setExpires(const String& val)`
		Specifies the date after which responses are considered stale.

Host	`String* getHost()`
	`const String* getHost() const`
	
	`void setHost(const String* val)`
	`void setHost(const String& val)`
		Specifies the Internet host and port number of the service for which the request is targeted.

Password	`String* getPassword()`
	`const String* getPassword() const`
	
	`void setPassword(const String* val)`
	`void setPassword(const String& val)`
		Specifies the password to use in username/password authentication.

Table 19: *Outgoing HTTP Client Attributes*

HTTP Attribute	Artix APIs	Description

`Pragma`	`String* getPragma()`
	`const String* getPragma() const`
	
	`void setPragma(const String* val)`
	`void setPragma(const String& val)`
		Specifies implementation-specific directives that might apply to any recipient along the request/response chain.

`Proxy-Authorization`	`String* getProxyAuthorization()`
	`const String* getProxyAuthorization() const`
	
	`void setProxyAuthorization(const String* val)`
	`void setProxyAuthorization(const String& val)`
		Specifies the credentials used to perform validation at a proxy server along the request/response chain. If the proxy uses username/password validation, this value is not used.

`ProxyAuthorizationType`	`String* getProxyAuthorizationType()`
	`String& getProxyAuthorizationType()`
	
	`void setProxyAuthorizationType(const String* val)`
	`void setProxyAuthorizationType(const String& val)`
		Specifies the type of authentication used by proxy servers along the request/response chain.

`ProxyPassword`	`String* getProxyPassword()`
	`const String* getProxyPassword() const`
	
	`void setProxyPassword(const String* val)`
	`void setProxyPassword(const String& val)`
		Specifies the password used by proxy servers for authentication if username/password authentication is in use.

`ProxyServer`	`String* getProxyServer()`
	`const String* getProxyServer() const`
	
	`void setProxyServer(const String* val)`
	`void setProxyServer(const String& val)`
		Specifies the URL of the proxy server, if one exists, along the request/response chain.
		Note: Artix does not support the existence of more than one proxy server along the request/response chain.

`ProxyUserName`	`String* getProxyUserName()`
	`const String* getProxyUserName() const`
	
	`void setProxyUserName(String val)`
		Specifies the username used by proxy servers for authentication if username/password authentication is in use.

`ReceiveTimeout`	`Int* getReceiveTimeout()`
	`const Int* getReceiveTimeout() const`
	
	`void setReceiveTimeout(const Int* val)`
	`void setReceiveTimeout(const Int& val)`
		Specifies the number of milliseconds the client will wait to receive a response from a server before timing out. The default is `3000`.

Table 19: *Outgoing HTTP Client Attributes*

HTTP Attribute	Artix APIs	Description

`Referer`	`String* getReferer()`
	`const String* getReferer() const`
	
	`void setReferer(const String* val)`
	`void setReferer(const String& val)`
		Specifies the entity that referred the client to the target server.

`Send-Timeout`	`Int* getSendTimeout()`
	`const Int* getSendTimeout() const`
	
	`void setSendTimeout(const Int* val)`
	`void setSendTimeout(const Int& val)`
		Specifies the number of milliseconds the client will continue trying to send a request to the server before timing out.

`ServerDate`	`String* getServerDate()`
	`const String* getServerDate() const`
	
	`void setServerDate(`
	`const String* val)`
	`void setServerDate(`
	`const String& val)`
		Specifies the time setting for the server. When this value is set, the client will use it as the base time from which to calculate message expiration. The client defaults to using its internal system clock.

`Trusted Root Certificate`	`String* getTrustedRootCertificates()`
	`const String*`
	`getTrustedRootCertificates() const`
	
	`void setTrustedRootCertificates(`
	`const String* val)`
	`void setTrustedRootCertificates(`
	`const String& val)`
		Specifies the full path to the PKCS12-encoded X509 certificate for the certificate authority.

`UserName`	`String* getUserName()`
	`const String* getUserName() const`
	
	`void setUserName(const String* val)`
	`void setUserName(const String& val)`
		Specifies the username used for authentication when the server uses username/password authentication.

`Use Secure Sockets`	`Boolean* getUseSecureSockets()`
	`const Boolean* getUseSecureSockets()`
	`const`
	
	`void setUseSecureSockets(`
	`const Boolean* val)`
	`void setUseSecureSockets(`
	`const Boolean& val)`
		Specifies the client wants to use a secure connection. Secure HTTP connections are also referred to as HTTPS.
		Valid values are `true` and `false`.
		Note: If the contract specifies HTTPS, this value is always `true`.

Incoming header

The client's incoming context, `HTTP_CLIENT_INCOMING_CONTEXTS`, is available in the client's reply context after a response from the server has been received by the transport layer. The values stored in this context are for informational purposes only. For example, if you need to check the MIME type of the data returned in the request, you would read it from the client's incoming context as shown in Example 109.

Example 109: *Reading the Content Type in an HTTP Client*

```
  // C++
1 #include <it_bus_pdk/context.h>
  #include <it_bus_pdk/context_attrs/http_conf_xsdTypes.h>
  ...
  IT_USING_NAMESPACE_STD

  using namespace IT_ContextAttributes;
  using namespace IT_Bus;
  ...
2 // make proxy invocation
  ...
3 ContextRegistry* context_registry =
      bus->get_context_registry();

  ContextCurrent& context_current =
      context_registry->get_current();

  // Obtain a pointer to the Request ContextContainer
  ContextContainer* context_container =
      context_current.reply_contexts();

4 // Obtain a reference to the context
  AnyType* info = context_container->get_context(
      IT_ContextAttributes::HTTP_CLIENT_INCOMING_CONTEXTS,
      true
  );

  // Cast the context into a clientType object
  clientType* http_client_config =
      dynamic_cast<clientType*> (info);

5 IT_Bus::String* content = http_client_config->getContentType();
```

The code in Example 109 does the following:

1. Includes the header files for the general context classes and for the HTTP client context type.
2. Makes an invocation on the proxy.
3. Gets the client's context registry.
4. Gets the client's incoming HTTP context from the reply context container.
5. Gets the value of the `ContentType` property.

Incoming client attributes

Table 20 shows the attributes that are valid in the incoming HTTP client context.

Table 20: *Incoming HTTP Client Attributes*

HTTP Attribute	Artix APIs	Description
Content-Encoding	String* getContentEncoding() const String* getContentEncoding() const	Specifies the type of special encoding, if any, the server used to package the response.
Content-Language	String* getContentLanguage() const String* getContentLanguage() const	Specifies the language the server used in writing the response. Valid language tags combine an ISO language code and an ISO country code separated by a hyphen. For example, en-US.
Content-Location	String* getContentLocation() const String* getContentLocation() const	Specifies the URL where the resource being sent in a response is located.
Content-Type	String* getContentType() const String* getContentType() const	Specifies the MIME type of the data in the response.
ETag	String* getETag() const String* getETag() const	Specifies the entity tag in the response header.
HTTPReply	String* getHTTPReply() const String* getHTTPReply() const	Specifies the type of reply being sent back by the server. For example, if a request is fulfilled a server will reply with OK.
HTTPReplyCode	Int* getHTTPReplyCode() const Int* getHTTPReplyCode() const	Specifies an integer code associated with the server's reply. For example, 200 means OK and 404 means Not Found.
Last-Modified	String* getLastModified() const String* getLastModified() const	Specifies the date and time at which the server believes a resource was last modified.
Proxy-Authenticate	String* getProxyAuthenticate() const String* getProxyAuthenticate() const	Specifies a challenge that indicates the authentication scheme and parameters applicable to the proxy for this Request-URI.
RedirectURL	String* getRedirectURL() const String* getRedirectURL() const	Specifies the URL to which client requests should be redirected. This is issued by a server when it is not appropriate for the request.

Table 20: *Incoming HTTP Client Attributes*

HTTP Attribute	Artix APIs	Description
ServerType	String* getServerType() const String* getServerType() const	Specifies the type of server responded to the client. Values take the form *program-name/version*.
WWW-Authenticate	String* getWWWAuthenticate() const String* getWWWAuthenticate() const	Specifies at least one challenge that indicates the authentication scheme(s) and parameters applicable to the Request-URI.

Server-side Configuration

HTTP servers have access to both the values being passed in the HTTP header of the outgoing response and the values received in the HTTP header of the request. The information for each header is stored in a separate context.

Outgoing header

On the server-side, the outgoing context, HTTP_SERVER_OUTGOING_CONTEXTS, is available in the server's reply context container. Any changes made to values in the outgoing context are placed in the reply's HTTP header and propagated to the client. For example, if you want to inform the client that it needs to redirect it's request to a different server, you could set the RedirectURL attribute in the server's outgoing context to the URL of an appropriate server. Example 110 shows the code for setting the RedirectURL attribute for a server.

Example 110:*Setting a Server's RedirectURL Attribute*

```
   // C++
1  #include <it_bus_pdk/context.h>
   #include <it_bus_pdk/context_attrs/http_conf_xsdTypes.h>
   ...
   IT_USING_NAMESPACE_STD

   using namespace IT_ContextAttributes;
   using namespace IT_Bus;
   ...
2  ContextRegistry* context_registry =
       bus->get_context_registry();

   ContextCurrent& context_current =
       context_registry->get_current();

   // Obtain a pointer to the Request ContextContainer
   ContextContainer* context_container =
       context_current.reply_contexts();
```

Example 110: *Setting a Server's RedirectURL Attribute*

3
```
// Obtain a reference to the context
AnyType* info = context_container->get_context(
    IT_ContextAttributes::HTTP_SERVER_OUTGOING_CONTEXTS,
    true
);

// Cast the context into a serverType object
serverType* http_server_config =
    dynamic_cast<serverType*> (info);
```

4
```
http_server_config->setRedirectURL("http://www.notme.org/askthi
   sguy");
```

The code in Example 110 does the following:

1. Includes the header files for the general context classes and for the HTTP server context type.
2. Gets the server's context registry.
3. Gets the server's outgoing HTTP context from the reply context container.
4. Sets the value of the `RedirectURL` property to the URL of the server that can satisfy the request.

Outgoing server attributes

Table 21 shows the attributes that are valid in the outgoing HTTP server context.

Table 21: *Outgoing HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description

`Cache-Control`	`String* getCacheControl()`
	`const String* getCacheControl() const`
	
	`void setCacheControl(const String* val)`
	`void setCacheControl(const String& val)`
		Specifies directives to caches along the request/response path.
		Valid values are:
		`no-cache`: caches must revalidate responses with the server. If response header fields are given, the restriction applies only to those header fields.
		`public`: any cache can store the response.
		`private`: public caches cannot store the response. If response header fields are given, the restriction applies only to those header fields.
		`no-store`: caches must not store any part of the response or the request.
		`no-transform`: caches must not modify the media type or the content location of a response.

Table 21: *Outgoing HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description
		must-revalidate: caches must revalidate responses that have expired with the server before the response can be used. **proxy-revalidate**: means the same as must-revalidate, but it can only be enforced on shared caches. You must set the public directive when using this directive. **max-age**: the max age, in seconds, of an acceptible response. **s-maxage**: means the same as max-age, but it can only be enforced on shared caches. When set it overides the value of max-age. You must use the proxy-revalidate directive when using this directive.
Content-Encoding	String* getContentEncoding() const String* getContentEncoding() const void setContextEncoding(const String* val) void setContextEncoding(const String& val)	Specifies the type of special encoding, if any, the server uses to package a response.
Content-Language	String* getContentLanguage() const String* getContentLanguage() const void setContentLanguage(const String* val) void setContentLanguage(const String& val)	Specifies the language used to write a response. Valid language tags combine an ISO language code and an ISO country code separated by a hyphen. For example, en-US.
Content-Location	String* getContentLocation() const String* getContentLocation() const void setContentLocation(const String* val) void setContentLocation(const String& val)	Specifies the URL where the resource being sent in a response is located.

Table 21: *Outgoing HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description

`Content-Type`	`String* getContentType()`
	`const String* getContentType() const`
	
	`void setContentType(const String* val)`
	`void setContentType(const String& val)`
		Specifies the MIME type of the data in the response.

`ETag`	`String* getETag()`
	`const String* getETag() const`
	
	`void setETag(const String* val)`
	`void setETag(const String& val)`
		Specifies the entity tag in the response header.

`Expires`	`String* getExpires()`
	`String& getExpires()`
	
	`void setExpires(const String* val)`
	`void setExpires(const String& val)`
		Specifies the date after which the response is considered stale.

`HonorKeepAlive`	`Boolean* getHonorKeepAlive()`
	`const Boolean* getHonorKeepAlive() const`
	
	`void setHonorKeepAlive(const Boolean* val)`
	`void setHonorKeepAlive(const Boolean& val)`
		Specifies if the server is going to honor a client's keep-alive request.

`HTTPReply`	`String* getHTTPReply()`
	`const String* getHTTPReply() const`
	
	`void setHTTPReply(const String* val)`
	`void setHTTPReply(const String& val)`
		Specifies the type of response the server is issuing. For example, if the request is fulfilled the server will reply with `OK`.

`HTTPReplyCode`	`Int* getHTTPReplyCode()`
	`const Int* getHTTPReplyCode() const`
	
	`void setHTTPReplyCode(const Int* val)`
	`void setHTTPReplyCode(const Int& val)`
		Specifies an integer code associated with the response. For example, `200` means `OK` and `404` means `Not Found`.

`Last-Modified`	`String* getLastModified()`
	`const String* getLastModified() const`
	
	`void setLastModified(const String* val)`
	`void setLastModified(const String& val)`
		Specifies the date and time at which the server believes a resource was last modified.

Table 21: *Outgoing HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description

`Pragma`	`String* getPragma()`
	`const String* getPragma() const`
	
	`void setPragma(const String* val)`
	`void setPragma(const String& val)`
		Specifies implementation-specific directives that might apply to any recipient along the request/response chain.

`Proxy-Authorization`	`String* getProxyAuthorization()`
	`const String* getProxyAuthorization() const`
	
	`void setProxyAuthorization( const String* val)`
	`void setProxyAuthorization( const String& val)`
		Specifies the credentials used to perform validation at a proxy server along the request/response chain. If the proxy uses username/password validation, this value is not used.

`ProxyAuthorizationType`	`String* getProxyAuthorizationType()`
	`const String* getProxyAuthorizationType() const`
	
	`void setProxyAuthorizationType( const String* val)`
	`void setProxyAuthorizationType( const String& val)`
		Specifies the type of authentication used by proxy servers along the request/response chain.

`ProxyPassword`	`String* getProxyPassword()`
	`const String* getProxyPassword() const`
	
	`void setProxyPassword( const String* val)`
	`void setProxyPassword( const String& val)`
		Specifies the password used by proxy servers for authentication if username/password authentication is in use.

`ProxyServer`	`String* getProxyServer()`
	`const String* getProxyServer() const`
	
	`void setProxyServer( const String* val)`
	`void setProxyServer( const String& val)`
		Specifies the URL of the proxy server, if one exists, along the request/response chain.
		Note: Artix does not support the existence of more than one proxy server along the request/response chain.

`ProxyUserName`	`String* getProxyUserName()`
	`const String* getProxyUserName() const`
	
	`void setProxyUserName( const String* val)`
	`void setProxyUserName( const String& val)`
		Specifies the username used by proxy servers for authentication if username/password authentication is in use.

Table 21: *Outgoing HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description

`Recieve-Timeout`	`Int* getRecieveTimeout()`
	`const Int* getRecieveTimeout() const`
	
	`void setRecieveTimeout(const Int* val)`
	`void setRecieveTimeout(const Int& val)`
		Specifies the number of milliseconds the server will wait to receive a request before timing out. The default is `3000`.

`RedirectURL`	`String* getRedirectURL()`
	`const String* getRedirectURL() const`
	
	`void setRedirectURL(const String* val)`
	`void setRedirectURL(const String& val)`
		Specifies the URL to which the request should be redirected.

`Send-Timeout`	`Int* getSendTimeout()`
	`const Int* getSendTimeout() const`
	
	`void setSendTimeout(const Int* val)`
	`void setSendTimeout(const Int& val)`
		Specifies the number of milliseconds the server will continue trying to send a response before timing out. The default is `3000`.

`ServerCertificate`	`String* getServerCertificate()`
	`const String* getServerCertificate() const`
	
	`void setServerCertificate(const String* val)`
	`void setServerCertificate(const String& val)`
		Specifies the full path to the X509 certificate issued by the certificate authority for the server.

`ServerCertificateChain`	`String* getServerCertificateChain()`
	`const String* getServerCertificateChain() const`
	
	`void setServerCertificateChain(const String* val)`
	`void setServerCertificateChain(const String& val)`
		Specifies the full path to the file containing all of the certificates in the chain.

`Server Type`	`String* getServerType()`
	`const String* getServerType() const`
	
	`void setServerType(const String* val)`
	`void setServerType(const String& val)`
		Specifies the type of server responded to the client. Values take the form *program-name/version*.

Table 21: *Outgoing HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description

`ServerPrivateKey`	`String* getServerPrivateKey()`
	`const String* getServerPrivateKey() const`
	
	`void setServerPrivateKey(const String* val)`
	`void setServerPrivateKey(const String& val)`
		Specifies the full path to the PKCS12-encoded private key that corresponds to the X509 certificate specified by `ServerCertificate`.

`ServerPrivateKeyPassword`	`String* getServerPrivateKeyPassword()`
	`const String* getServerPrivateKeyPassword() const`
	
	`void getServerPrivateKeyPassword(const String* val)`
	`void getServerPrivateKeyPassword(const String& val)`
		Specifies the password used to decrypt the PKCS12-encoded private key.

`Trusted Root Certificate`	`String* getTrustedRootCertificates()`
	`const String* getTrustedRootCertificates() const`
	
	`void setTrustedRootCertificates(const String* val)`
	`void setTrustedRootCertificates(const String& val)`
		Specifies the full path to the PKCS12-encoded X509 certificate for the certificate authority.

`UseSecureSockets`	`Boolean* getUseSecureSockets()`
	`const Boolean* getUseSecureSockets() const`
	
	`void setUseSecureSockets(const Boolean* val)`
	`void setUseSecureSockets(const Boolean& val)`
		Specifies the server wants to use a secure connection. Secure HTTP connections are also referred to as HTTPS.
		Note: If the contract specifies HTTPS, this value is always `true`.

`WWW-Authenticate`	`String* getWWWAuthenticate()`
	`const String* getWWWAuthenticate() const`
	
	`void setWWWAunthenticate(const String* val)`
	`void setWWWAunthenticate(const String& val)`
		Specifies at least one challenge that indicates the authentication scheme(s) and parameters applicable to the Request-URI.

Incoming header

The server's incoming context, `HTTP_SERVER_INCOMING_CONTEXTS`, is available in the server's request context container after a request from client has been received by the transport layer. The values stored in this context are for informational purposes only. For example, if you need to check the MIME type of the data the client can accept in the response, you would read it from the server's incoming context as shown in Example 111.

Example 111: *Reading the Accept Attribute in an HTTP Server*

```
  // C++
1 #include <it_bus_pdk/context.h>
  #include <it_bus_pdk/context_attrs/http_conf_xsdTypes.h>
  ...
  IT_USING_NAMESPACE_STD

  using namespace IT_ContextAttributes;
  using namespace IT_Bus;
  ...
2 ContextRegistry* context_registry =
      bus->get_context_registry();

  ContextCurrent& context_current =
      context_registry->get_current();

  // Obtain a pointer to the Request ContextContainer
  ContextContainer* context_container =
      context_current.request_contexts();

3 // Obtain a reference to the context
  AnyType* info = context_container->get_context(
      IT_ContextAttributes::HTTP_SERVER_INCOMING_CONTEXTS, true
  );

  // Cast the context into a serverType object
  serverType* http_server_config =
      dynamic_cast<serverType*> (info);

4 IT_Bus::String* content = http_server_config->getAccept();
```

The code in Example 111 does the following:

1. Includes the header files for the general context classes and for the HTTP server context type.
2. Gets the server's context registry.
3. Gets the server's incoming HTTP context from the request context container.
4. Gets the value of the `Accept` property.

Incoming server attributes

Table 22 shows the attributes that are valid in the incoming HTTP server context.

Table 22: *Incoming HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description
`Accept`	`String* getAccept()` `const String* getAccept() const`	Specifies the MIME types the client can handle in a response.
`Accept-Encoding`	`String* getAcceptEncoding()` `const String* getAcceptEncoding() const`	Specifies the types of content encoding the client can handle in a response. This property typically refers to compression mechanisms.
`Accept-Language`	`String* getAcceptLanguage()` `const String* getAcceptLanguage() const`	Specifies the language preferred by the client. Valid language tags combine an ISO language code and an ISO country code separated by a hyphen. For example, `en-US`.
`Authorization`	`String* getAuthorization()` `const String* getAuthorization() const`	Specifies the credentials that will be used by the server to authorize requests from the client.
`AuthorizationType`	`String* getAuthorizationType()` `const String* getAuthorizationType() const`	Specifies the name of the authentication scheme in use.
`AutoRedirect`	`Boolean* getAutoRedirect()` `const Boolean* getAutoRedirect() const`	Specifies whether the server should automatically redirect the request.
`BrowserType`	`String* getBrowserType()` `const String* getBrowserType() const`	Specifies information about the browser from which the request originates. This property is also know as the user-agent.
`Certificate Issuer`	`String* getCertificateIssuer()` `const String* getCertificateIssuer() const`	Specifies the value stored in the `Issuer` field of the client's X509 certificate.
`Certificate Key Size`	`Int* getCertificateKeySize()` `const Int* getCertificateKeySize() const`	Specifies the size, in bytes, of the public key included in the client's x509 certificate.
`Certificate Valid Not After`	`String* getCertificateNotAfter()` `const String* getCertificateNotAfter() const`	Specifies the date and time after which the client's X509 certificate is invalid.

Table 22: *Incoming HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description
`Certificate Valid Not Before`	`String* getCertificateNotBefore()` `const String* getCertificateNotBefore() const`	Specifies the date and time before which the client's X509 certificate is invalid.
`Certificate Subject`	`String* getCertificateSubject()` `const String* getCertificateSubject() const`	Specifies the value of the `Subject` field in the client's X509 certificate.
`Connection`	`String* getConnection()` `const String* getConnection() const`	Specifies whether a connection is to be kept open after each request/response transaction.
`Cookie`	`String* getCookie()` `const String* getCookie() const`	Specifies a static cookie that is sent along with a request. **Note:** According to the HTTP 1.1 specification, HTTP cookies must contain US-ASCII characters.
`Host`	`String* getHost()` `const String* getHost() const`	Specifies the Internet host and port number of the resource being requested.
`HTTPVersion`	`String* getHTTPVersion()` `const String* getHTTPVersion() const`	Specifies the version of the HTTP transport in use. Currently, this is always set to 1.1.
`If-Modified-Since`	`String* getIfModifiedSince()` `const String* getIfModifiedSince() const`	If the requested resource has not been modified since the time specified, the server should issue a 304 (not modified) response without any message body.
`Method`	`String* getMethod()` `const String* getMethod() const`	Specifies the value of the `METHOD` token sent in the request. Valid values and their meanings are given in the HTTP 1.1 specification.
`Passwrod`	`String* getPassword()` `const String* getPassword() const`	Specifies the password the client wishes to use for authentication.
`Proxy-Authenticate`	`String* getProxyAuthenticate()` `const String* getProxyAuthenticate() const`	Specifies a challenge that indicates the authentication scheme and parameters applicable to the proxy for this Request-URI.
`Referer`	`String* getReferer()` `const String* getReferer() const`	Specifies the entity that referred the client.

Table 22: *Incoming HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description
`URL`	`String* getURL()` `const String* getURL() const`	Specifies the value of the Request-URI sent in the request. The valid values for this property are described in the HTTP 1.1 specification.
`Username`	`String* getUserName()` `const String* getUserName() const`	Specifies the username the client wishes to use for authentication.

Setting the Server's Endpoint URL

Because the server's endpoint URL must be known before the transport layer is initialized by the bus, you must use the specialized configuration context to set it. For more information on using the configuration context see "Getting a Context Instance" on page 161.

Side effects

A side effect of setting the server's endpoint URL using contexts is that the following configuration variables:

```
# Artix Configuration File
policies:soap:server_address_mode_policy:publish_hostname
policies:at_http:server_address_mode_policy:publish_hostname
```

are ignored. The endpoint addresses advertised by the WSDL publish service will reflect the values set in the configuration context, not the values set in the configuration file.

Getting the property

To access the HTTP endpoint URL property for an HTTP server, obtain a configuration context container (using `get_configuration_context()`) and then get the `HTTP_SERVER_OUTGOING_CONTEXTS` context. You are returned an `IT_ContextAttributes::serverType` object that has two relevant methods:

- `setURL()` sets a `String` representing the URL of the server.
- `getURL()` returns a `String` representing the URL of the server.

Server main function

Example 112 shows sample code from a server main function, which shows how to initialize `http-conf:serverType` configuration context data.

Example 112: *Server Main Function Setting a Configuration Context*

```
// C++

#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_cal/iostream.h>

// Include header files related to the soap context
1 #include <it_bus_pdk/context.h>
2 #include <it_bus_pdk/context_attrs/http_conf_xsdTypes.h>

IT_USING_NAMESPACE_STD

using namespace IT_ContextAttributes;
using namespace IT_Bus;

int
main(int argc, char* argv[])
{
    try
    {
        IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);

3       IT_Bus::QName service_name(
            "",
            "SOAPService",
            "http://www.iona.com/hello_world_soap_http"
        );

4       ContextRegistry* context_registry =
            bus->get_context_registry();

5       ContextContainer * context_container =
            context_registry->get_configuration_context(
              service_name,
              "SoapPort",
              true
           );

        // Obtain a reference to the context
6       AnyType* info = context_container->get_context(
           IT_ContextAttributes::HTTP_SERVER_OUTGOING_CONTEXTS,
            true
        );

        // Cast the context into a serverType object
7       serverType* http_server_config =
            dynamic_cast<serverType*> (info);

        // Modify the endpoint URL
8       http_server_config->setURL("http://localhost:63278");
        ...
```

Example 112:*Server Main Function Setting a Configuration Context*

```
        GreeterImpl servant(bus);
        bus->register_servant(
                    servant,
                    "../../etc/hello_world.wsdl",
                    service_name
             );
    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Error : Unexpected error occured!"
             << endl << e.message()
             << endl;
        return -1;
    }
    return 0;
}
```

The preceding code example can be explained as follows:

1. The `it_bus_pdk/context.h` header file contains the declarations of the following classes:
   - `IT_Bus::ContextRegistry`,
   - `IT_Bus::ContextContainer`,
   - `IT_Bus::ContextCurrent`.
2. The `http_conf_xsdTypes.h` header declares the context data types generated from the `http-conf` schema.
3. This `service_name` is the QName of the SOAP service featured in the `hello_world_soap_http` demonstration (in `samples/basic/hello_world_soap_http`).
4. Obtain a reference to the `IT_Bus::ContextRegistry` object, which is used to register contexts with the Bus.
5. The `IT_Bus::ContextContainer` object returned by `get_configuration_context()` holds configuration data that is used exclusively by the specified endpoint (that is, the `SoapPort` port in the `SOAPService` service).
6. The `IT_Bus::ContextContainer::get_context()` function is called with its second parameter set to `true`, indicating that a context with that name should be created if none already exists.
7. The `IT_Bus::AnyType` class is the base type for all complex types in Artix. In this case, you can cast the `AnyType` instance, `info`, to its derived type, `serverType`.
8. You can now modify the URL used by the `SoapPort` port by calling the `setURL()` function.

Setting CORBA Attributes

The CORBA transport does not support programmatic configuration, nor does it provide access to any of the settings that are used to establish the connection. Artix does, however, provide access to the CORBA principal by way of the context mechanism. The CORBA principal is manipulated as a `String` by the contexts.

For details of how to use the CORBA principal in Artix, consult the *Artix Security Guide*.

Setting WebSphere MQ Attributes

When working with WebSphere MQ, your applications can access information about the WebSphere MQ connection that is in use and information contained in the WebSphere MQ message descriptor. The MQ connection attributes context contains information about the queues and queue managers that your application uses for sending and receiving messages. On the client-side, you can set this information on a per-invocation basis. The MQ message attributes context allows you to inspect and set a number of the properties stored in the WebSphere MQ message descriptor.

Working with Connection Attributes

The WebSphere MQ transport provides information about the queues to which your application send and receives messages. This information is stored in the MQ connection attributes context and is accessed using the `MQ_CONNECTION_ATTRIBUTES` constant. The data is returned in an `MQConnectionAttributesType` object. Table 23 describes the attributes stored in the MQ connection attributes context.

Table 23: *MQ Connection Attributes Context Properties*

Attribute	Artix APIs	Description

AliasQueueName	`String* getAliasQueueName()`
	`const String* getAliasQueueName() const`
	
	`void setAliasQueueName(const String* val)`
	`void setAliasQueueName(const String& val)`
		Specifies the remote queue to which a server will put replies if its queue manager is not on the same host as the client's local queue manager.

ConnectionName	`String* getConnectionName()`
	`const String* getConnectionName() const`
	
	`void setConnecitonName(const String* val)`
	`void setConnecitonName(const String& val)`
		Specifies the name of the connection by which the adapter connects to the queue.

ModelQueueName	`String* getModelQueueName()`
	`const String* getModelQueueName() const`
	
	`void setModelQueueName(const String* val)`
	`void setModelQueueName(const String& val)`
		Specifies the name of the queue to be used as a model for creating dynamic queues.

QueueManager	`String* getQueueManager()`
	`const String* getQueueManager() const`
	
	`void setQueueManager(const String* val)`
	`void setQueueManager(const String& val)`
		Specifies the name of the queue manager.

Table 23: *MQ Connection Attributes Context Properties*

Attribute	Artix APIs	Description

QueueName	`String* getQueueName()`
	`const String* getQueueName() const`
	
	`void setQueueName(const String* val)`
	`void setQueueName(const String& val)`
		Specifies the name of the message queue.

ReplyQueueManager	`String* getReplyQueueManager()`
	`const String* getReplyQueueManager() const`
	
	`void setReplyQueueManager( const String* val)`
	`void setReplyQueueManager( const String& val)`
		Specifies the name of the reply queue manager. This setting is ignored by WebSphere MQ servers when the client specifies the `ReplyToQMgr` in the request message's message descriptor.

ReplyQueueName	`String* getReplyQueueName()`
	`const String* getReplyQueueName() const`
	
	`void setReplyQueueName(const String* val)`
	`void setReplyQueueName(const String& val)`
		Specifies the name of the queue where response messages are received. This setting is ignored by WebSphere MQ servers when the client specifies the `ReplyToQ` in the request message's message descriptor.

Transactional	`TransactionType* getTransactional()`
	`const TransactionType* getTransactional() const`
	
	`void setTransactional( const TransactionType* val)`
	`void setTransactional( const TransactionType& val)`
		Specifies how messages participate in transactions and what role WebSphere MQ plays in the transactions. For information on setting Transactional see "Setting the Transactional attribute" on page 242.

On the client-side you can control the connection to which requests are directed by setting the MQ connection attributes in the client's request context before each invocation. The connection attributes are returned to the defaults specified in the client's contract after each invocation.

Example

Example 113 shows code for specifying the queue and queue manager to use when making a request.

Example 113: *Setting the Client's QueueManager and QueueName*

```
// C++
1 #include <it_bus_pdk/context.h>
#include <it_bus_pdk/context_attrs/mq_xsdTypes.h>
...
```

Example 113: *Setting the Client's QueueManager and QueueName*

```
IT_USING_NAMESPACE_STD

using namespace IT_ContextAttributes;
using namespace IT_Bus;
...
2 ContextRegistry* context_registry =
    bus->get_context_registry();

ContextCurrent& context_current =
    context_registry->get_current();

// Obtain a pointer to the Request ContextContainer
ContextContainer* context_container =
    context_current.request_contexts();

3 // Obtain a reference to the context
AnyType* info = context_container->get_context(
    IT_ContextAttributes::MQ_CONNECTION_ATTRIBUTES,
    true
);

// Cast the context into a MQConnectionAttributesType object
MQConnectionAttributesType* mq_client_config =
    dynamic_cast<MQConnectionAttributesType*> (info);

4 mq_client_config->setQueueManager("Bloggy");
mq_client_config->setQueueName("TalkBack");

// make proxy invocations
...
```

The code in Example 113 does the following:

1. Includes the header files for the general context classes and for the MQ connection attributes context type.
2. Gets the client's context registry.
3. Gets the client's MQ connection attributes context from the request context container.
4. Sets the queue manager attribute and the queue name attribute.

Note: On the server-side you cannot change any of the connection attributes programmatically.

Setting the Transactional attribute

The transactional attribute is set using a `transactionType` object. `transactionType` is a WSDL enumeration whose values are described in Table 24.

Table 24: *MQ Transactional Values*

Value	Artix API for Setting	Description
`none`	`setTransactional(transactionType::none)`	The messages are not part of a transaction. No rollback actions will be taken if errors occur.
`internal`	`setTransactional(transactionType::internal)`	The messages are part of a transaction with WebSphere MQ serving as the transaction manager.
`xa`	`setTransactional(transactionType::xa)`	The messages are part of a transaction with WebSphere MQ serving as the resource manager.

Example 114 shows code for setting a client's connection to use XA style transactionality for a request.

Example 114: *Setting the Client's Transactionality Attribute*

```
   // C++
1  #include <it_bus_pdk/context.h>
   #include <it_bus_pdk/context_attrs/mq_xsdTypes.h>
   ...
   IT_USING_NAMESPACE_STD

   using namespace IT_ContextAttributes;
   using namespace IT_Bus;
   ...
2  ContextRegistry* context_registry =
       bus->get_context_registry();

   ContextCurrent& context_current =
       context_registry->get_current();

   // Obtain a pointer to the Request ContextContainer
   ContextContainer* context_container =
       context_current.request_contexts();

3  // Obtain a reference to the context
   AnyType* info = context_container->get_context(
       IT_ContextAttributes::MQ_CONNECTION_ATTRIBUTES,
       true
   );

   // Cast the context into a MQConnectionAttributesType object
   MQConnectionAttributesType* mq_client_config =
       dynamic_cast<MQConnectionAttributesType*> (info);
```

Example 114: *Setting the Client's Transactionality Attribute*

4
```
mq_client_config->setTransactional(transactionType::xa);

// make proxy invocations
...
```

The code in Example 113 does the following:

1. Includes the header files for the general context classes and for the MQ connection attributes context type.
2. Gets the client's context registry.
3. Gets the client's MQ connection attributes context from the request context container.
4. Sets the MQ transaction type to XA.

For more information about working with Artix enumerated types, see "Deriving Simple Types by Restriction" on page 299.

Working with MQ Message Descriptor Attributes

The Artix WebSphere MQ transport splits its MQ message descriptor attributes between two contexts, as follows:

- MQ incoming message attributes.
- MQ outgoing message attributes.

MQ incoming message attributes

One context, accessed using the `MQ_INCOMING_MESSAGE_ATTRIBUTES constant`, contains the MQ message descriptor attributes for the last message received by the application. For a client, this means that it contains the attributes for the last response received from the server and the context is accessed through the client's reply context container. For a server, this means that the incoming message attributes context contains the descriptor attributes for the request being processed and it is accessed through the server's request context container. The incoming message properties can be read at any point in the processing of the message once the transport layer has passed it to the messaging chain.

MQ outgoing message attributes

The second context, accessed using `MQ_OUTGOING_MESSAGE_ATTRIBUTES`, allows you to set the values of the attributes in the MQ message descriptor for the next message being sent across the wire. For clients, this means that it affects the values of the next request being made and the context is accessed through the client's request context. For server's, this means that the outgoing message attributes context affects the values of the current response's MQ message descriptor and it is accessed through the server's reply context container. You can set the values of the outgoing message attributes at any point in an application's message chain before it the message is handed off to the transport layer.

MQ message attributes

Both the incoming message attributes context and the outgoing message attributes context are returned using as an `MQMessageAttributesType` object. Table 25 describes the attributes stored in the MQ message attributes context.

Table 25: *MQ Message Attributes Context Properties*

Attribute	Artix APIs	Description

AccountingToken	`String* getAccountingToken()`
	`const String* getAccountingToken() const`
	
	`void setAccountingToken(const String* val)`
	`void setAccountingToken(const String& val)`
		Specifies the value for the MQ message descriptor's `AccountingToken` field.

ApplicationData	`String* getApplicationData()`
	`const String* getApplicationData() const`
	
	`void setApplicationData(const String* val)`
	`void setApplicationData(const String& val)`
		Specifies any application-specific information that needs to be set in the message descriptor.

ApplicationIdData	`String* getApplicationIdData()`
	`const String* getApplicationIdData() const`
	
	`void setApplicationIdData( const String* val)`
	`void setApplicationIdData( const String& val)`
		Specifies the value of the MQ message descriptor's `ApplIdentityData` field. It is only valid for MQ clients.

ApplicationOriginData	`String* getApplicationOriginData()`
	`const String* getApplicationOriginData() const`
	
	`void setApplicationOriginData( const String* val)`
	`void setApplicationOriginData( const String& val)`
		Specifies the value of the MQ message descriptor's `ApplOriginData` field.

BackoutCount	`Int* getBackoutCount()` `const Int* getBackoutCount() const`	Returns the number of times the message has been previously returned by the `MQGET` call as part of a unit of work, and subsequently backed out.

Convert	`Boolean* isConvert()`
	`const Boolean* isConvert() const`
	
	`void setConvert(const Boolean* val)`
	`void setConvert(const Boolean& val)`
		Specifies if the messages in the queue needs to be converted to the system's native encoding.

Table 25: *MQ Message Attributes Context Properties*

Attribute	Artix APIs	Description

CorrelationID	`Base64Binary* getCorrelationID()`
	`const Base64Binary* getCorrelationID() const`
	
	`void setCorrelationID(const Base64Binary* val)`
	`void setCorrelationID(const Base64Binary& val)`
		Specifies the value for the MQ message descriptor's `CorrelId` field.

CorrelationStyle	`correlationStyleType* getCorrelationStyle()`
	`const correlationStyleType* getCorrelationStyle() const`
	
	`void setCorrelationStyle(const correlationStyleType* val)`
	`void setCorrelationStyle(const correlationStyleType& val)`
		Specifies how WebSphere MQ matches both the message identifier and the correlation identifier to select a particular message to be retrieved from the queue. For information on how to set CorrelationStyle, see "Setting the CorrelationStyle attribute" on page 246.

Delivery	`deliveryType* getDelivery()`
	`const deliveryType* getDelivery() const`
	
	`void setDelivery(const deliveryType* val)`
	`void setDelivery(const deliveryType& val)`
		Specifies the value of the MQ message descriptor's `Persistence` field. For information on setting Delivery, see "Setting the Delivery attribute" on page 247.

Format	`formatType* getFormat()`
	`const formatType* getFormat() const`
	
	`void setFormat(const formatType* val)`
	`void setFormat(const formatType& val)`
		Specifies the value of the MQ message descriptor's `Format` field. For information on setting Format, see "Setting the Format attribute" on page 248.

MessageID	`String* getMessageID()`
	`const String* getMessageID() const`
	
	`void setMessageID(const String* val)`
	`void setMessageID(const String& val)`
		Specifies the value for the MQ message descriptor's `MsgId` field.

ReportOption	`reportOptionType* getReportOption()`
	`const reportOptionType* getReportOption() const`
	
	`void setReportOption(const reportOptionType* val)`
	`void setReportOption(const reportOptionType& val)`
		Specifies the value of the MQ message descriptor's `Report` field. For information on setting ReportOption, see "Setting the ReportOption attribute" on page 250.

Table 25: *MQ Message Attributes Context Properties*

Attribute	Artix APIs	Description

UserIdentifier	`String* getUserIdentifier()`
	`const String* getUserIdentifier() const`
	
	`void setUserIdentifier(const String* val)`
	`void setUserIdentifier(const String& val)`
		Specifies the value for the MQ message descriptor's `UserIdentifier` field.

Setting the CorrelationStyle attribute

The CorrelationStyle attribute is set using a `correlationStyleType` object. `correlationStyleType` is a WSDL enumeration whose values are described in Table 26.

Table 26: *CorrelationStyle Values*

Value	Artix API for Setting	Description
`messageId`	`correlationStyleType cs("messageId");` `context->setCorrelationStyle(cs);`	Use the message ID as the value for the message's `CorrelId`.
`correlationId`	`correlationStyleType cs("correlationId");` `context->setCorrelationStyle(cs);`	Use the message's CorrelationId as the value for the message's `CorrelId`.
`messageId copy`	`correlationStyleType cs("messageId copy");` `context->setCorrelationStyle(cs);`	Use the message ID as the value for the message's `MsgId`.

Example 115 shows code for setting a request message descriptor's CorrelationStyle message Id.

Example 115: *Setting the Client's CorrelationStyle Attribute*

```
   // C++
1  #include <it_bus_pdk/context.h>
   #include <it_bus_pdk/context_attrs/mq_xsdTypes.h>
   ...
   IT_USING_NAMESPACE_STD

   using namespace IT_ContextAttributes;
   using namespace IT_Bus;
   ...
2  ContextRegistry* context_registry =
       bus->get_context_registry();

   ContextCurrent& context_current =
       context_registry->get_current();

   // Obtain a pointer to the Request ContextContainer
   ContextContainer* context_container =
       context_current.request_contexts();
```

Example 115: *Setting the Client's CorrelationStyle Attribute*

```
3 // Obtain a reference to the context
  AnyType* info = context_container->get_context(
      IT_ContextAttributes::MQ_OUTGOING_MESSAGE_ATTRIBUTES,
      true
  );

  // Cast the context into a MQMessageAttributesType object
  MQMessageAttributesType* mq_msg_config =
      dynamic_cast<MQMessageAttributesType*> (info);

4 correlationStyleType cs("messageId");
  mq_msg_config->setCorrelationStyle(cs);

  // make proxy invocations
  ...
```

The code in Example 115 does the following:

1. Includes the header files for the general context classes and for the MQ message attributes context type.
2. Gets the client's context registry.
3. Gets the client's MQ outgoing message attributes context from the request context container.
4. Sets the correlation style to `messageId`.

Setting the Delivery attribute

The Delivery attribute is set using a `deliveryType` object. `deliveryType` is a WSDL enumeration whose values are described in Table 27.

Table 27: *Delivery Values*

Value	Artix API for Setting	Description
`persistent`	`deliveryType delivery_t("persistent");` `context->setDelivery(delivery_t)`	Sets the `Persistence` field to `MQPER_PERSISTENT`.
`not persistent`	`deliveryType delivery_t("not persistent");` `context->setDelivery(delivery_t);`	Sets the `Persistence` field to `MQPER_NOT_PERSISTENT`.

Example 116 shows code for setting a request message descriptor's `Persistence` field to `MQPER_PERSISTENT`.

Example 116: *Setting the Client's Delivery Attribute*

```
  // C++
1 #include <it_bus_pdk/context.h>
  #include <it_bus_pdk/context_attrs/mq_xsdTypes.h>
  ...
  IT_USING_NAMESPACE_STD

  using namespace IT_ContextAttributes;
  using namespace IT_Bus;
  ...
```

Example 116: *Setting the Client's Delivery Attribute*

```
2  ContextRegistry* context_registry =
       bus->get_context_registry();

   ContextCurrent& context_current =
       context_registry->get_current();

   // Obtain a pointer to the Request ContextContainer
   ContextContainer* context_container =
       context_current.request_contexts();

3  // Obtain a reference to the context
   AnyType* info = context_container->get_context(
       IT_ContextAttributes::MQ_OUTGOING_MESSAGE_ATTRIBUTES,
       true
   );

   // Cast the context into a MQMessageAttributesType object
   MQMessageAttributesType* mq_msg_config =
       dynamic_cast<MQMessageAttributesType*> (info);

4  deliveryType delivery_t("persistent");
   mq_msg_config->setDelivery(delivery_t);

   // make proxy invocations
   ...
```

The code in Example 116 does the following:

1. Includes the header files for the general context classes and for the MQ message attributes context type.
2. Gets the client's context registry.
3. Gets the client's MQ outgoing message attributes context from the request context container.
4. Sets the delivery type to `persistent`.

Setting the Format attribute

The Format attribute is set using a `formatType` object. `formatType` is a WSDL enumeration whose values are described in Table 28.

Table 28: *Format Values*

Value	Artix API for Setting	Description
none	`formatType format("none");` `context->setFormat(format);`	Sets the `Format` field to `MQFMT_NONE`.
string	`formatType format("string");` `context->setFormat(format);`	Sets the `Format` field to `MQFMT_STRING`.
unicode	`formatType format("unicode");` `context->setFormat(format);`	Sets the `Format` field to `MQFMT_STRING`.
event	`formatType format("event");` `context->setFormat(format);`	Sets the `Format` field to `MQFMT_EVENT`.

Table 28: *Format Values*

Value	Artix API for Setting	Description
programmable command	`formatType format("programmable command");` *`context`*`->setFormat(format);`	Sets the `Format` field to `MQFMT_PCF`.
ims	`formatType format("ims");` *`context`*`->setFormat(format);`	Sets the `Format` field to `MQFMT_IMS`.
ims_var_string	`formatType format("ims_var_string");` *`context`*`->setFormat(format);`	Sets the `Format` field to `MQFMT_IMS_VAR_STRING`.

Example 117 shows code for setting a request message descriptor's `Format` field to `MQFMT_STRING`.

Example 117: *Setting the Client's Format Attribute*

```
   // C++
1  #include <it_bus_pdk/context.h>
   #include <it_bus_pdk/context_attrs/mq_xsdTypes.h>
   ...
   IT_USING_NAMESPACE_STD

   using namespace IT_ContextAttributes;
   using namespace IT_Bus;
   ...
2  ContextRegistry* context_registry =
       bus->get_context_registry();

   ContextCurrent& context_current =
       context_registry->get_current();

   // Obtain a pointer to the Request ContextContainer
   ContextContainer* context_container =
       context_current.request_contexts();

3  // Obtain a reference to the context
   AnyType* info = context_container->get_context(
       IT_ContextAttributes::MQ_OUTGOING_MESSAGE_ATTRIBUTES,
       true
   );

   // Cast the context into a MQMessageAttributesType object
   MQMessageAttributesType* mq_msg_config =
       dynamic_cast<MQMessageAttributesType*> (info);

4  formatType format("string");
   mq_msg_config->setFormat(format);

   // make proxy invocations
   ...
```

The code in Example 117 does the following:

1. Includes the header files for the general context classes and for the MQ message attributes context type.
2. Gets the client's context registry.

3. Gets the client's MQ outgoing message attributes context from the request context container.
4. Sets the message format to `string`.

Setting the ReportOption attribute

The `ReportOption` attribute is set using a `reportOptionType` object. `ReportOptionType` is a WSDL enumeration whose values are described in Table 29.

Table 29: *ReportOption Values*

Value	Artix API for Setting	Description
coa	`reportOptionType report_option("coa");` `context->setReportOption(report_option)`	Set the message descriptor's `Report` field to `MQRO_COA`.
cod	`reportOptionType report_option("cod");` `context->setReportOption(report_option)`	Set the message descriptor's `Report` field to `MQRO_COD`.
exception	`reportOptionType report_option("exception");` `context->setReportOption(report_option)`	Set the message descriptor's `Report` field to `MQRO_EXCEPTION`.
expiration	`reportOptionType report_option("expiration");` `context->setReportOption(report_option)`	Set the message descriptor's `Report` field to `MQRO_EXPIRATION`.
discard	`reportOptionType report_option("discard");` `context->setReportOption(report_option)`	Set the message descriptor's `Report` field to `MQRO_DISCARD_MSG`.

Example 118 shows code for setting a request message descriptor's `Report` field to `MQRO_DISCARD_MSG`.

Example 118:*Setting the Client's ReportOption Attribute*

```
   // C++
1  #include <it_bus_pdk/context.h>
   #include <it_bus_pdk/context_attrs/mq_xsdTypes.h>
   ...
   IT_USING_NAMESPACE_STD

   using namespace IT_ContextAttributes;
   using namespace IT_Bus;
   ...
2  ContextRegistry* context_registry =
       bus->get_context_registry();

   ContextCurrent& context_current =
       context_registry->get_current();

   // Obtain a pointer to the Request ContextContainer
   ContextContainer* context_container =
       context_current.request_contexts();
```

Example 118: *Setting the Client's ReportOption Attribute*

3
```
// Obtain a reference to the context
AnyType* info = context_container->get_context(
    IT_ContextAttributes::MQ_OUTGOING_MESSAGE_ATTRIBUTES,
    true
);

// Cast the context into a MQMessageAttributesType object
MQMessageAttributesType* mq_msg_config =
    dynamic_cast<MQMessageAttributesType*> (info);
```

4
```
reportOptionType report_option("discard");
mq_msg_config->setReportOption(report_option)

// make proxy invocations
...
```

The code in Example 118 does the following:

1. Includes the header files for the general context classes and for the MQ message attributes context type.
2. Gets the client's context registry.
3. Gets the client's MQ outgoing message attributes context from the request context container.
4. Sets the report option to `discard`.

Setting FTP Attributes

The attributes used to configure an FTP connection are split into four contexts:

- one for setting the policies used to connect to the FTP daemon.
- one for setting the credentials to use when connecting to the FTP daemon.
- one for setting the naming scheme implementation to use for Artix clients.
- one for setting the naming scheme implementation to use for Artix servers.

These settings are all controlled through the special configuration context that is made available before Artix registers any user level code with the bus. For more information on using the configuration context see "Getting a Context Instance" on page 161.

Artix clients can dynamically set the scan interval used by the FTP transport. and can dynamically adjust the length of time they will wait for a response before timing out.

Setting FTP Connection Policies

When setting the FTP connection policies you access them using the `FTP_CONNECTION_POLICY` tag. The FTP connection policy context information is returned as a `IT_ContextAttributes::ConnectionPolicyType` object. All of the connection policies are valid when set in the configuration context. In addition, Artix clients can set the scan interval policy and the receive timeout policy in their request contexts.

Setting the connection mode

The FTP connection mode is set using a `ConnectModeType` object. `ConnectModeType` is an enumeration whose values are described in Table 30.

Table 30: *ConnectionMode Values*

Value	Artix API for Setting	Description
`active`	`ConnectModeType connect_mode("active");` `context->setconnectMode(connect_mode);`	Specifies that Artix controls the connection to the FTPD.
`passive`	`ConnectModeType connect_mode("passive");` `context->setconnectMode(connect_mode);`	Specifies that the FTPD controls the connection.

Example 119 shows code for setting the connection mode to `passive`.

Example 119: *Setting the FTP Connection Mode*

```
  // C++
1 #include <it_bus_pdk/context.h>
  #include <it_bus_pdk/context_attrs/ftp_context_xsdTypes.h>
  ...
  IT_USING_NAMESPACE_STD

  using namespace IT_ContextAttributes;
  using namespace IT_Bus;
  ...
2 ContextRegistry* context_registry =
      bus->get_context_registry();

3 QName service_qname
      = new QName("http://www.iona.com/ftp_example",
  "FTPService");

4 ContextContainer* context_container =
      context_registry.get_configuration_context(
          service_qname,
          "FTPPort",
          true
      );

5 // Obtain a reference to the context
  AnyType* info = context_container->get_context(
      IT_ContextAttributes::FTP_CONNECTION_POLICY,
      true
  );
```

Example 119: *Setting the FTP Connection Mode*

```
// Cast the context into a ConnectionPolicyType object
ConnectionPolicyType* ftp_config =
    dynamic_cast<ConnectionPolicyType*> (info);

6 ConnectModeType connect_mode("passive");
ftp_config->setconnectMode(connect_mode);

// make proxy invocations
...
```

The code in Example 119 does the following:

1. Includes the header files for the general context classes and for the FTP connection policy type.
2. Gets the client's context registry.
3. Set the name of an FTP service defined in the WSDL contract. For example, you might define an FTP service like the following:

```
<wsdl:definitions name="HelloWorld"
    targetNamespace="http://www.iona.com/ftp_example"  ... >
    ...
    <wsdl:service name="FTPService">
        <wsdl:port binding="tns:Greeter_FTPBinding"
                   name="FTPPort">
            <ftp:port host="FTPHost" port="3210" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

4. The configuration context is specific to the endpoint defined by the service, `FTPService`, and the port, `FTPPort`.
5. Gets the client's FTP connection policy context from the configuration context container.
6. Sets the FTP connection mode to `passive`.

Setting the connection timeout

The FTP connection time out determines the number of seconds Artix will spend in attempting to connect to the FTPD before timing out. It is set using `setconnectTimeout()`. The value is specified as an integer as shown in Example 120.

Example 120: *Setting the Connection Timeout Policy*

```
// C++
AnyType* info = context_container->get_context(
    IT_ContextAttributes::FTP_CONNECTION_POLICY,
    true
);
ConnectionPolicyType* ftp_config =
    dynamic_cast<ConnectionPolicyType*> (info);

ftp_config.setconnectTimeout(10);
```

Setting the scan interval

The scan interval determines the number of seconds that Artix waits before rescaning the remote message repository for new messages. In addition to being settable in the configuration context, the scan interval can also be set by Artix clients using the request context.

It is set using `setscanInterval()`. The value is specified as an integer, as shown in Example 121.

Example 121: *Setting the Scan Interval in a Client*

```
// C++
AnyType* info = context_container->get_context(
    IT_ContextAttributes::FTP_CONNECTION_POLICY,
    true
);
ConnectionPolicyType* ftp_config =
    dynamic_cast<ConnectionPolicyType*> (info);

ftp_config.setscanInterval(3);

// Make invocation on proxy
```

Setting the receive timeout

The receive timeout determines the number of seconds that an Artix client waits for a response before throwing a timeout exception. In addition to being settable in the configuration context, the receive timeout can also be set by Artix clients using the request context.

It is set using `setrecieveTimeout()`. The value is specified as an integer as shown in Example 122.

Example 122: *Setting the Receive Timeout in a Client*

```
// C++
AnyType* info = context_container->get_context(
    IT_ContextAttributes::FTP_CONNECTION_POLICY,
    true
);
ConnectionPolicyType* ftp_config =
    dynamic_cast<ConnectionPolicyType*> (info);

ftp_config.setreceiveTimeout(60);

// Make invocation on proxy
```

Setting the Connection Credentials

FTP servers require you to connect using a username and password. These are set using the FTP connection credentials property.

Because the username and password used to connect to the FTP server must be known before the transport is initialized, you need to set the property in the special configuration context that is made available before Artix registers any user level code with the bus. For more information on using the configuration context see "Getting a Context Instance" on page 161.

Setting the FTP connection credentials

To set the FTP connection credentials property, use the `FTP_CREDENTIALS` tag. You are returned a `CredentialsType` object that has four member functions:

- `setname()` sets a `String` representing the username used when connecting to the FTP server.
- `getname()` returns a `String` representing the username used when connecting to the FTP server.
- `setpassword()` sets a `String` representing the password used when connecting to the FTP server.
- `getpassword()` returns a `String` representing the password used when connecting to the FTP server.

Example

Example 123 shows how to set the FTP connection credentials properties on an Artix FTP client.

Example 123:*Setting the FTP Connection Mode*

```
  // C++
1 #include <it_bus_pdk/context.h>
  #include <it_bus_pdk/context_attrs/ftp_context_xsdTypes.h>
  ...
  IT_USING_NAMESPACE_STD

  using namespace IT_ContextAttributes;
  using namespace IT_Bus;
  ...
2 ContextRegistry* context_registry =
      bus->get_context_registry();

3 QName service_qname
      = new QName("http://www.iona.com/ftp_example",
  "FTPService");

4 ContextContainer* context_container =
      context_registry.get_configuration_context(
          service_qname,
          "FTPPort",
          true
      );

5 // Obtain a reference to the context
  AnyType* info = context_container->get_context(
      IT_ContextAttributes::FTP_CREDENTIALS,
      true
  );
```

Example 123: *Setting the FTP Connection Mode*

```
// Cast the context into a CredentialsType object
CredentialsType* creds =
    dynamic_cast<CredentialsType*> (info);

6 creds->setname("george");
creds->setpassword("bosco");

// make proxy invocations
...
```

The code in Example 123 does the following:

1. Includes the header files for the general context classes and for the FTP credentials policy type.
2. Gets the client's context registry.
3. Set the name of an FTP service defined in the WSDL contract. For example, you might define an FTP service like the following:

```
<wsdl:definitions name="HelloWorld"
 targetNamespace="http://www.iona.com/ftp_example"  ... >
    ...
    <wsdl:service name="FTPService">
        <wsdl:port binding="tns:Greeter_FTPBinding"
                   name="FTPPort">
            <ftp:port host="FTPHost" port="3210" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

4. The configuration context is specific to the endpoint defined by the service, `FTPService`, and the port, `FTPPort`.
5. Gets the client's FTP credentials policy context from the configuration context container.
6. Sets the username and password for the FTP connection.

Setting the Naming Policies

The FTP naming policies determine how Artix names the files created for the messages sent over the FTP transport and how Artix cleans up files on the remote datastore. These behaviors are controlled by a set of Java classes that you can implement to meet specific needs. Artix also provides default implementations.

For details, see the "Using FTP" section in the "*Transports*" chapter of ***Bindings and Transports, C++ Runtime*** guide.

Setting i18n Attributes

Artix has two contexts to configure codeset conversion when using the i18n interceptor. One context configures the client and the other configures the server. The i18n interceptor is used when working in an environment where codeset conversion is required, but the transports in use do not support it. It is a message-level interceptor and is invoked just before the transport layer is handed the message.

The i18n interceptor can also be set up using port extensors in your application's contract. For information on setting up the i18n interceptor using port extensors see the chapter on services in Designing Artix Solutions.

Configuring Artix to use the i18n interceptor

Before your application can use the i18n interceptor for code conversion you must configure the Artix bus to load the required plug-ins and add the interceptor to the appropriate message interceptor lists. To configure your application to use the i18n interceptor do the following:

1. If your application includes a service proxy that needs to use codeset conversion, add `"I18nInterceptorFactory"` to the `binding:artix:client_message_interceptor_list` variable for your application.
2. If your application includes a service that needs to use codeset conversion, add `"I18nInterceptorFactory"` to the `binding:artix:server_message_interceptor_list` variable for your application.
3. Add `"i18n_interceptor"` to the list of plug-ins to load in the `orb_plugins` variable for your application.

For more information on configuring Artix see ***Configuring and Deploying Artix Solutions***.

Setting up i18n on a client

In a client the only attributes in the i18n context that alter how the i18n interceptor works are the client local codeset and the client outbound codeset in the client's request context. The client inbound codeset defaults to the value of the outbound codeset and the client-side interceptor does not read its value from the context.

To configure a client for codeset conversion using the i18n interceptor do the following:

1. Get the client's message context.
2. Get the i18n client request context.
3. Set the local codeset property.
4. Set the outbound codeset property.

Example 124 shows the code for configuring a client for codeset conversion.

Example 124:*Client i18n Properties*

```
  // C++
  #include <it_bus_pdk/context.h>
  #include <it_bus_pdk/context_attrs/i18n_context_xsdTypes.h>
  ...
  IT_USING_NAMESPACE_STD

  using namespace IT_ContextAttributes;
  using namespace IT_Bus;
  ...
1 ContextRegistry* context_registry =
      bus->get_context_registry();
  ContextCurrent& context_current =
      context_registry->get_current();
  ContextContainer* context_container =
      context_current.request_contexts();

2 AnyType* info = context_container->get_context(
      IT_ContextAttributes::I18N_INTERCEPTOR_CLIENT_QNAME,
      true
  );
  ClientConfiguration* i18n_config =
      dynamic_cast<ClientConfiguration*> (info);

3 i18n_config->setLocalCodeSet("Latin-1");
4 i18n_config->setOutboundCodeSet("UTF-16");
```

Setting up i18n on a server

In a server the only attributes in the i18n context that alter how the i18n interceptor works are the server local codeset and the server outbound codeset in the server's reply context. The server-side interceptor does not read the server inbound codeset from the context.

To configure a server for codeset conversion using the i18n interceptor do the following:

1. Get the server's message context.
2. Get the i18n server reply context.
3. Set the local codeset property.
4. Set the outbound codeset property.

[Example 125](#) shows the code for configuring a server for codeset conversion.

Example 125: *Server i18n Properties*

```
// C++
#include <it_bus_pdk/context.h>
#include <it_bus_pdk/context_attrs/i18n_context_xsdTypes.h>
...
IT_USING_NAMESPACE_STD

using namespace IT_ContextAttributes;
using namespace IT_Bus;
...
1 ContextRegistry* context_registry =
      bus->get_context_registry();
  ContextCurrent& context_current =
      context_registry->get_current();
  ContextContainer* context_container =
      context_current.request_contexts();

2 AnyType* info = context_container->get_context(
      IT_ContextAttributes::I18N_INTERCEPTOR_SERVER_QNAME,
      true
  );
  ServerConfiguration* i18n_config_srvr =
      dynamic_cast<ServerConfiguration*> (info);

3 i18n_config_srvr->setLocalCodeSet("Latin-1");
4 i18n_config_srvr->setOutboundCodeSet("UTF-16");
```

Setting WS-A and WS-RM Attributes

The WS-ReliableMessaging (WS-RM) specification describes an interoperable protocol that provides message delivery guarantees between a source and a destination. The protocol is layered above SOAP.

In addition to supporting oneway and synchronous two-way calls, the WS-RM protocol can also work with *message sequences*. Delivery guarantees can be applied to message sequences—for example, you can require that every message in a message sequence gets delivered to its destination.

Enabling reliable messaging

In order to enable reliable messaging, you must update the Artix configuration file. For details of how to configure WS-RM, see *Configuring and Deploying Artix Solutions*.

Demonstration code

A demonstration of the WS-ReliableMessaging feature is provided in the following directory:

ArtixInstallDir`/samples/advanced/wsrm`

Setting the WS-A ReplyTo Endpoint

The WS-Addressing (WS-A) message exchange pattern is a basic pre-requisite for WS-ReliableMessaging. Essentially, the message exchange pattern provides the basic infrastructure for setting up a two-way stream of messages between a source and a destination. When this pattern is enabled, Artix sends a SOAP header that contains a `wsa:To` element and a `wsa:ReplyTo` element to the server. The Artix core then sends request messages to the endpoint specified in the `wsa:To` element and receives reply messages *asynchronously* at the endpoint specified in the `wsa:ReplyTo` element.

The `IT_Bus::WSAConfigurationContext` context enables you to specify the `wsa:ReplyTo` URI programmatically on the client side.

WS-A configuration context scope

When you register a WS-A configuration context instance, it is valid for one proxy and one proxy only. The first proxy on which you invoke an operation will adopt the programmed settings. The settings will *not* apply to any proxies that you create subsequently.

Setting the ReplyTo endpoint for a client proxy

Example 126 shows how to set the WS-Addressing ReplyTo endpoint on a client proxy.

Example 126:*Setting the WS-A ReplyTo Endpoint on a Client Proxy*

```
   // C++
1  #include <it_bus_pdk/context_attrs/context_constants.h>
   #include <it_bus_pdk/context_attrs/wsa_config_context.h>

2  ContextContainer* request_container =
   m_bus->get_pdk_bus()->get_context_registry()->get_current().req
      uest_contexts();

   ClientProxy proxy;

3  WSAConfigurationContext* wsa_config_context
       = new WSAConfigurationContext();
4  wsa_config_context->set_wsa_replyto_uri(
       "http://localhost:0/WSAContextClient/ContextReplyTo"
   );

5  request_container->add_context(
       IT_ContextAttributes::WSA_CONFIGURATION_CONTEXT,
       *wsa_config_context
   );

6  proxy.hello_world();
```

The preceding code example can be explained as follows:

1. Includes the header files for the general context classes and the WS-Addressing configuration context type.
2. Gets the request context container.
3. Create an `IT_Bus::WSAConfigurationContext` instance to hold the WS-RM attributes.
4. Call the `set_wsa_replyto_uri()` function to specify the ReplyTo URI. The address in this URI can be set as follows:
   - *Fixed host and port*—where you specify the name of the client host explicitly and you choose an explicit IP port number (non-zero).
   - *Dynamically allocated address*—where you specify the placeholder address, `localhost:0`, and leave it up to the operating system to allocate an IP port number. Artix replaces `localhost` with the name of the client host. The client then transmits the dynamically allocated address to the server inside a SOAP header (using the `wsa:replyTo` element).
5. When you have finished adding WS-Addressing attributes on the WS-Addressing configuration context instance, add the context to the request context container.
6. The first proxy on which you invoke an operation adopts the WS-Addressing settings and clears the context again. The settings then apply to all subsequent operation calls made using this proxy. Other proxy instances are *not* affected by the WS-Addressing settings.

Alternative way to set the ReplyTo endpoint

An alternative way of setting the ReplyTo endpoint is by setting the value of the endpoint reference explicitly. Example 127 shows how to set the WS-Addressing ReplyTo endpoint on a client proxy, using the `IT_Bus::WSAConfigurationContext::set_wsa_2005_replyto_epr()` function.

Example 127: *Alternative Way to Set the WS-A ReplyTo Endpoint*

```
// C++
#include <it_bus_pdk/context_attrs/context_constants.h>
#include <it_bus_pdk/context_attrs/wsa_config_context.h>

ContextContainer* request_container =
m_bus->get_pdk_bus()->get_context_registry()->get_current().request_contexts();

ClientProxy proxy;

WSAConfigurationContext* wsa_config_context = new WSAConfigurationContext();
WS_Addressing::EndpointReferenceType reply_to_epr;
1 reply_to_epr.setAddress("http://localhost:0/WSAContextClient/ContextReplyTo");
2 wsa_config_context->set_wsa_2005_replyto_epr(reply_to_epr);


request_container->add_context(
```

Example 127: *Alternative Way to Set the WS-A ReplyTo Endpoint*

```
    IT_ContextAttributes::WSA_CONFIGURATION_CONTEXT,
    *wsa_config_context
);

proxy.hello_world();
```

The preceding code example can be explained as follows:

1. Pass the URL address to the `WS_Addressing::EndpointReferenceType` constructor. Instead of setting the endpoint address directly as an URL string, you must first wrap the URL address in an endpoint reference type.
2. Set the ReplyTo endpoint by calling the `EndpointReferenceType::set_wsa_2005_replyto_epr()` function.

Setting WS-RM Attributes

The basic settings for enabling WS-RM must be specified in the Artix configuration file (see *Configuring and Deploying Artix Solutions*). It is possible, however, to override some of the settings by programming the WS-RM configuration context, as described here.

RM sources and RM destinations

The reliable messaging protocol is based on the concept of an RM channel, which transmits messages in one direction only. Each channel consists of an *RM source* (where messages originate) and an *RM destination* (where messages arrive).

For each client-server association, there are two basic ways of organizing RM channels, as follows:

- *One-way association*—sends oneway messages from a client to a server. The association consists of a single channel, with an RM source on the client side and an RM destination on the server side.
- *Two-way association*—sends messages in both directions, between a client and a server. This association consists of two channels, where the client and the server each have an RM source and an RM destination.

WS-RM configuration context scope

When you register a WS-RM configuration context instance, it is valid for one proxy and one proxy only. The first proxy on which you invoke an operation will adopt the programmed settings. The settings will *not* apply to any proxies that you create subsequently.

Moreover, WS-RM attributes are by definition applicable either to an RM source or to an RM destination (either of which can occur in a client or in a server). This contrasts with other kinds of transport attribute, which are applicable either to a client or to a server.

Setting WS-RM attributes on a client proxy

Example 128 shows the general approach to setting WS-RM attributes that affect a particular client proxy instance, `proxy`.

Example 128:*Setting WS-RM Attributes on a Client Proxy*

```
  // C++
1 #include <it_bus_pdk/context_attrs/context_constants.h>
  #include <it_bus_pdk/context_attrs/wsrm_config_context.h>
  ...
  IT_USING_NAMESPACE_STD

  using namespace IT_ContextAttributes;
  using namespace IT_Bus;
  ...
  ContextContainer* request_container =
2 m_bus->get_pdk_bus()->get_context_registry()->get_current().request_contexts();

  ClientProxy proxy;

3 WSRMConfigurationContext* wsrm_config_context
      = new WSRMConfigurationContext();

4 // Set WS-RM attributes here!
  ...

5 request_container->add_context(
      IT_ContextAttributes::WSRM_CONFIGURATION_CONTEXT,
      *wsrm_config_context
  );

6 proxy.hello_world();
```

The preceding code example can be explained as follows:

1. Includes the header files for the general context classes and the WS-RM configuration context type.
2. Gets the request context container.
3. Create an `IT_Bus::WSRMConfigurationContext` instance to hold the WS-RM attributes.
4. You can set any of the client-side WS-RM attributes at this point in the code (not shown).
5. When you have finished adding WS-RM attributes on the WS-RM configuration context instance, add the context to the request context container.
6. The first proxy on which you invoke an operation adopts the WS-RM settings and clears the context again. The settings then apply to all subsequent operation calls made using this proxy. Other proxy instances are *not* affected by the WS-RM settings.

Setting WS-RM attributes in a servant

On the server side, you can set RM source attributes by modifying the attributes in a WS-RM reply context *before* the service sends its first reply message to a particular client. RM destination attributes, on the other hand, cannot be modified by programming on the server side.

Example 129 shows the general approach to setting WS-RM attributes in a servant (that is, in the implementation of an operation).

Example 129:*Setting WS-RM Attributes in a Servant*

```
  // C++
1 #include <it_bus_pdk/context_attrs/context_constants.h>
  #include <it_bus_pdk/context_attrs/wsrm_config_context.h>
  ...
  IT_USING_NAMESPACE_STD

  using namespace IT_ContextAttributes;
  using namespace IT_Bus;
  ...
2 // Obtain a pointer to the reply ContextContainer
  ContextContainer* reply_container =
  m_bus->get_context_registry()->get_current().reply_contexts();

3 WSRMConfigurationContext* wsrm_config_context
      = new WSRMConfigurationContext();

4 // Set WS-RM source attributes here!
  ...

5 reply_container->add_context(
      IT_ContextAttributes::WSRM_CONFIGURATION_CONTEXT,
      *wsrm_config_context
  );
```

The preceding code example can be explained as follows:

1. Includes the header files for the general context classes and the WS-RM configuration context type.
2. Gets the reply context container.
3. Create an `IT_Bus::WSRMConfigurationContext` instance to hold the server-side WS-RM attributes.
4. You can set RM source attributes at this point in the code (not shown).
5. When you have finished adding WS-RM attributes on the WS-RM configuration context instance, add the context to the request context container.

Programmable WS-RM source attributes

You can set the following WS-RM source attributes programmatically:

- WS-RM acknowledgement URI.
- Base re-transmission interval.

- Disable exponential backoff.
- Max unacknowledged messages threshold.
- Maximum retransmission attempts.
- Maximum messages per sequence.
- Per-thread sequence scope.

WS-RM acknowledgement URI

The WS-RM acknowledgement URI specifies the endpoint where the WS-RM source receives acknowledgement messages. In a SOAP header, this attribute is represented by the `wsrm:AcksTo` element. The default is the standard WS-A anonymous URI:

`http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous`

There are three alternative methods for specifying the WS-RM acknowledgement URI, as follows:

- You can set the WS-RM acknowledgement URI explicitly by inserting the following code fragment into Example 128 on page 263 or into Example 129 on page 264:

```
// C++
WSRMConfigurationContext* wsrm_config_context = new
   WSRMConfigurationContext();

AnyURI acksto_url(
    "http://localhost:0/WSASource/DemoContextAcksTo"
);
WS_Addressing_2004::AttributedURI acks_to_uri(acksto_url);

wsrm_config_context->set_wsrm_acknowledgement_uri(
    acks_to_uri
);
```

- A proxy that is used to make two-way invocations can be configured so that its decoupled reply-to endpoint, `wsa:replyTo` (which receives application responses), also receives WS-RM acknowledgements. For example:

```
// C++
WSRMConfigurationContext* wsrm_config_context = new WSRMConfigurationContext();

wsrm_config_context->use_wsa_replyto_endpoint_for_wsrm_acknowledgement();
```

- A service that is used to make two-way invocations can be configured so that the server endpoint (which receives application requests) can also be used to receive WS-RM acknowledgements (in other words, acts as a `wsrm:acksTo` endpoint for the reverse WS-RM channel). For example:

```
// C++
WSRMConfigurationContext* wsrm_config_context = new WSRMConfigurationContext();

wsrm_config_context->use_server_endpoint_for_wsrm_acknowledgement();
```

The order of preference for choosing a `wsrm:acksTo` endpoint is as follows:

1. If the WS-RM source endpoint is explicitly configured (through the Artix configuration file or by programming) to use a non-anonymous `wsrm:acksTo` endpoint, then use it.
2. The second preference depends on whether the setting is being made on the client side or on the server side, as follows:
   - On the client side, you can configure the WS-RM source endpoint to use the `wsa:replyTo` endpoint as the `wsrm:acksTo` endpoint.
   - On the server side, you can configure the WS-RM source endpoint to use the server endpoint as the `wsrm:acksTo` endpoint.
3. If neither 1 or 2 is specified, use an anonymous `wsrm:acksTo` endpoint.

Base re-transmission interval

The base re-transmission interval specifies the interval at which a WS-RM source re-transmits a message that has not yet been acknowledged. The default is 2000 milliseconds.

You can set the base re-transmission interval by inserting the following code fragment into Example 128 on page 263 or into Example 129 on page 264:

```
// C++
WSRMConfigurationContext* wsrm_config_context = new WSRMConfigurationContext();
wsrm_config_context->set_base_retransmission_interval(3000);
```

Disable exponential backoff

This attribute specifies whether or not successive re-transmission attempts for an unacknowledged message are done at exponential time intervals. If `true`, the re-transmission is done at the base re-transmission interval; if `false`, the re-transmission is exponentially backed off. The default is `false`.

You can disable the exponential backoff algorithm by inserting the following code fragment into Example 128 on page 263 or into Example 129 on page 264

```
// C++
WSRMConfigurationContext* wsrm_config_context = new WSRMConfigurationContext();
wsrm_config_context->disable_exponential_backoff();
```

Max unacknowledged messages threshold

The *maximum unacknowledged messages threshold* specifies the maximum number of unacknowledged messages tolerated at the WS-RM source. When the threshold is exceeded, the WS-RM source ceases sending messages (and the application thread remains blocked) until the number of unacknowledged messages falls below the threshold again. The default is -1 (which represents no limit on the number of unacknowledged messages). You can set the maximum unacknowledged messages threshold by inserting the following code fragment into Example 128 on page 263 or into Example 129 on page 264:

```
// C++
WSRMConfigurationContext* wsrm_config_context = new WSRMConfigurationContext();
wsrm_config_context->set_max_unacked_messages_threshold(50);
```

Maximum retransmission attempts

The *maximum retransmission attempts* specifies the maximum number of times a WS-RM source will attempt to retransmit an unacknowledged message. If the number of retransmission attempts reaches this threshold, the WS-RM source sends a `wsrm:SequenceTerminated` fault to the peer WS-RM destination, and then closes the session. Any subsequent attempt to send message on this session will result in an `IT_Bus::Exception` being thrown. The default is -1 (which represents no limit on the number of retransmission attempts).

You can set the maximum retransmission attempts threshold by inserting the following code fragment into Example 128 on page 263 or into Example 129 on page 264:

```
// C++
WSRMConfigurationContext* wsrm_config_context = new WSRMConfigurationContext();
wsrm_config_context->set_max_retransmission_attempts(8);
```

Maximum messages per sequence

The *maximum messages per sequence* determines the maximum number of user messages allowed in a WS-RM sequence. The default is unlimited, which is appropriate for most cases.

If a limit is set using this property, the RM source creates a new sequence whenever the specified limit is reached and all acknowledgements for the previously sent messages have been received.

You can set the maximum number of messages per sequence by inserting the following code fragment into Example 128 on page 263 or into Example 129 on page 264:

```
// C++
WSRMConfigurationContext* wsrm_config_context = new WSRMConfigurationContext();
wsrm_config_context->set_max_messages_per_sequence(1);
```

Per-thread sequence scope

When a WS-RM source is invoked concurrently, the WS-RM session is normally shared by all threads (this is the default). When the *per-thread sequence scope* policy is enabled, however, the WS-RM source endpoint transparently creates a distinct

WS-RM sequence session for each invoking thread. This eliminates the possibility of message IDs being allocated to messages indeterminately in the presence of multiple threads. In other words, all the messages sent by a particular thread would be allocated message IDs in increasing order. When the WS-RM source closes, it closes all of the open WS-RM sequence sessions.

The default value of this policy is `false` (disabled).

You can enable the per-thread sequence scope policy by inserting the following code fragment into Example 128 on page 263 or into Example 129 on page 264:

```
// C++
WSRMConfigurationContext* wsrm_config_context = new WSRMConfigurationContext();
wsrm_config_context->enable_per_thread_sequence_scope();
```

Programmable WS-RM destination attributes

You can set the following WS-RM destination attribute programmatically:

- Acknowledgement interval.
- Delivery assurance policies.

Acknowledgement interval

The acknowledgement interval specifies the time interval at which the WS-RM destination sends asynchronous acknowledgements. The default is 3000 milliseconds.

You can set the acknowledgement interval by inserting the following code fragment into Example 128 on page 263:

```
// C++
WSRMConfigurationContext* wsrm_config_context = new WSRMConfigurationContext();
wsrm_config_context->set_acknowledgement_interval(2500);
```

Note: It is *not* possible to set the acknowledgement interval programmatically on the server side. On the server side, the acknowledgement interval can be set only in configuration.

Delivery assurance policies

A WS-RM destination can be configured to have the following kinds of delivery assurance policies:

- `ExactlyOnceInOrder`—the WS-RM destination delivers the messages to the application destination exactly once, in increasing order of the WS-RM message ID. Calls to the application destination are, therefore, serialized. This is the default policy value.

- `ExactlyOnceConcurrent`—the WS-RM destination delivers the messages to the application destination exactly once, but not in order. Instead of a serialized delivery of the messages, as in the case of `ExactlyOnceInOrder`, the WS-RM destination delivers the messages in the context of the WS-RM workqueue threads, so the ordering is not guaranteed. What is guaranteed, however, is that for a message, `n`, being delivered, all messages in the range `1` to `n` are received and acknowledged by the WS-RM destination.
- `ExactlyOnceReceivedOrder`—the WS-RM destination delivers messages to the application destination exactly-once, as soon as they are received from the underlying transport. The WS-RM destination makes no attempt to ensure either that the messages are delivered in the order of message ID or that all the previous messages have been received/acknowledged. The benefit of this policy is that it avoids a context-switch during dispatch in the RM layer and also the messages are not stored in the in-memory undelivered messages map.

The default value of this policy is `ExactlyOnceInOrder`.

You can set the delivery assurance policy by inserting the following code fragment into Example 128 on page 263:

```
// C++
WSRMConfigurationContext* wsrm_config_context = new WSRMConfigurationContext();
wsrm_config_context->set_acknowledgement_interval(2500);
```

Artix Data Types

This chapter presents the XML schema data types supported by Artix and describes how these data types map to C++. The Artix WSDL-to-C++ mapping conforms to the official OMG specification, http://www.omg.org/cgi-bin/doc?mars/06-06-38.

Including and Importing Schema Definitions

Artix supports the including and importing of schema definitions, using the `<include/>` and `<import/>` schema tags. These tags enable you to insert definitions from external files or resources into the scope of a `schema` element. The essential difference including and importing is this:

- Including brings in definitions that belong to the *same* target namespace as the enclosing `schema` element, whereas
- Importing brings in definitions that belong to a *different* target namespace from the enclosing `schema` element.

xsd:include syntax

The include directive has the following syntax:

```
<include
  schemaLocation = "anyURI"
/>
```

The referenced schema, given by *anyURI*, must either belong to the same target namespace as the enclosing schema or not belong to any target namespace at all. If the referenced schema does not belong to any target namespace, it is automatically adopted into the enclosing schema's namespace when it is included.

xsd:import syntax

The import directive has the following syntax:

```
<import
  namespace = "namespaceAnyURI"
  schemaLocation = "schemaAnyURI"
/>
```

The imported definitions must belong to the *namespaceAnyURI* target namespace. If *namespaceAnyURI* is blank or remains unspecified, the imported schema definitions are unqualified.

Example

Example 130 shows an example of an XML schema that includes another XML schema.

Example 130: *Example of a Schema that Includes Another Schema*

```
<definitions
   targetNamespace="http://schemas.iona.com/tests/schema_parser
   "
xmlns:tns="http://schemas.iona.com/tests/schema_parser"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

    <types>
        <schema
targetNamespace="http://schemas.iona.com/tests/schema_parser"
            xmlns="http://www.w3.org/2001/XMLSchema">

           <include schemaLocation="included.xsd"/>

            <complexType name="IncludingSequence">
                <sequence>
                    <element
                        name="includedSeq"
                        type="tns:IncludedSequence"/>
                </sequence>
            </complexType>

        </schema>
    </types>
<...>
```

Example 131 shows the contents of the included schema file, `included.xsd`.

Example 131: *Example of an Included Schema*

```
<schema
targetNamespace="http://schemas.iona.com/tests/schema_parser"
            xmlns="http://www.w3.org/2001/XMLSchema">
    <!-- Included type definitions -->
    <complexType name="IncludedSequence">
        <sequence>
            <element name="varInt" type="int"/>
            <element name="varString" type="string"/>
        </sequence>
    </complexType>
</schema>
```

Simple Types

This section describes the WSDL-to-C++ mapping for simple types. Simple types are defined within an XML schema and they are subject to the restriction that they cannot contain elements and they cannot carry any attributes.

This section contains the following types:

- Atomic Types
- String Type
- NormalizedString and Token Types
- QName Type
- Date and Time Types
- Duration Type
- Decimal Type
- Integer Types
- Binary Types
- Deriving Simple Types by Restriction
- List Type
- Union Type
- Holder Types
- Unsupported Simple Types

Atomic Types

For unambiguous, portable type resolution, a number of data types are defined in the Artix foundation classes, specified in `it_bus/types.h`.

Table of atomic types

The atomic types are:

Table 31: *Simple Schema Type to Simple Bus Type Mapping*

Schema Type	Bus Type
xsd:boolean	IT_Bus::Boolean
xsd:byte	IT_Bus::Byte
xsd:unsignedByte	IT_Bus::UByte
xsd:short	IT_Bus::Short
xsd:unsignedShort	IT_Bus::UShort
xsd:int	IT_Bus::Int
xsd:unsignedInt	IT_Bus::UInt
xsd:long	IT_Bus::Long
xsd:unsignedLong	IT_Bus::ULong

Table 31: *Simple Schema Type to Simple Bus Type Mapping*

Schema Type	Bus Type
xsd:float	IT_Bus::Float
xsd:double	IT_Bus::Double
xsd:string	IT_Bus::String
xsd:normalizedString	IT_Bus::NormalizedString
xsd:token	IT_Bus::Token
xsd:language	IT_Bus::Language
xsd:NMTOKEN	IT_Bus::NMToken
xsd:NMTOKENS	IT_Bus::NMTokens
xsd:Name	IT_Bus::Name
xsd:NCName	IT_Bus::NCName
xsd:ID	IT_Bus::ID
xsd:QName	IT_Bus::QName *(SOAP only)*
xsd:duration	IT_Bus::Duration
xsd:dateTime	IT_Bus::DateTime
xsd:date	IT_Bus::Date
xsd:time	IT_Bus::Time
xsd:gDay	IT_Bus::GDay
xsd:gMonth	IT_Bus::GMonth
xsd:gMonthDay	IT_Bus::GMonthDay
xsd:gYear	IT_Bus::GYear
xsd:gYearMonth	IT_Bus::GYearMonth
xsd:decimal	IT_Bus::Decimal
xsd:integer	IT_Bus::Integer
xsd:positiveInteger	IT_Bus::PositiveInteger
xsd:negativeInteger	IT_Bus::NegativeInteger
xsd:nonPositiveInteger	IT_Bus::NonPositiveInteger
xsd:nonNegativeInteger	IT_Bus::NonNegativeInteger
xsd:base64Binary	IT_Bus::BinaryBuffer
xsd:hexBinary	IT_Bus::BinaryBuffer

String Type

The xsd:string type maps to IT_Bus::String, which is typedef'ed in it_bus/ustring.h to IT_Bus::IT_UString class. For a full definition of IT_Bus::String, see it_bus/ustring.h.

IT_Bus::String class

The IT_Bus::String class is modelled on the standard ANSI string class. Hence, the IT_Bus::String class overloads the + and += operators for concatenation, the [] operator for indexing characters, and the ==, !=, >, <, >=, <= operators for comparisons.

String iterator class

The corresponding string iterator class is IT_Bus::String::iterator.

Example

The following C++ example shows how to perform some basic string manipulation with IT_Bus::String:

```
// C++
IT_Bus::String s = "A C++ ANSI string."
s += " And here is some string concatenation."

// Now convert to a C style string.
// (Note: s retains ownership of the memory)
const char *p = s.c_str();
```

Internationalization

The IT_Bus::String class supports the use of international characters. When using international characters, you should configure your Artix application to use a particular code set by editing the Artix domain configuration file, artix.cfg. The configuration details depend on the type of Artix binding, as follows:

- SOAP binding—set the plugins:soap:encoding configuration variable.
- CORBA binding—set the plugins:codeset:char:ncs, plugins:codeset:char:ccs, plugins:codeset:wchar:ncs, and plugins:codeset:wchar:ccs configuration variables.

For more details about configuring internationalization, see the "Using Artix with International Codesets" chapter of the *Configuring and Deploying Artix Solutions* document.

Encoding arguments

Some of the IT_Bus::String functions take an optional string argument, encoding, that lets you specify a character set encoding for the string.

The encoding argument must be a standard IANA character set name. For example, Table 32 shows some of commonly used IANA character set names:

Table 32: *IANA Character Set Names*

IANA Name	Description
US-ASCII	7-bit ASCII for US English.
ISO-8859-1	Western European languages.
UTF-8	Byte oriented transformation of Unicode.
UTF-16	Double-byte oriented transformation of 4-byte Unicode.
Shift_JIS	Japanese DOS & Windows.
EUC-JP	Japanese adaptation of generic EUC scheme, used in UNIX.
EUC-CN	Chinese adaptation of generic EUC scheme, used in UNIX.
ISO-2022-JP	Japanese adaptation of generic ISO 2022 encoding scheme.
ISO-2022-CN	Chinese adaptation of generic ISO 2022 encoding scheme.
BIG5	Big Five is a character set developed by a consortium of five companies in Taiwan in 1984.

Artix supports all of the character sets defined in International Components for Unicode (ICU) 2.6. For a full listing of supported character sets, see http://www-124.ibm.com/icu/index.html (part of the IBM open source project http://oss.software.ibm.com).

Constructors

The IT_Bus::String class defines a default constructor and non-default constructors to initialize a string using narrow and wide characters, as follows:

- Narrow character constructors.
- 16-bit character constructor.
- wchar_t character constructor.

Narrow character constructors

Example 132 shows three different constructors that can be used to initialize an `IT_UString` with a narrow character string.

Example 132: *Narrow Character Constructors*

```
IT_UString(
    const char*           str,
    size_t                n  = npos,
    const char*           encoding = 0,
    IT_ExceptionHandler& eh = IT_EXCEPTION_HANDLER
);
IT_UString(
    size_t                n,
    char                  c,
    const char*           encoding = 0,
    IT_ExceptionHandler& eh = IT_EXCEPTION_HANDLER
);
IT_UString(
    const IT_String&     s,
    size_t                pos = 0,
    size_t                n   = npos,
    const char*           encoding = 0,
    IT_ExceptionHandler& eh  = IT_EXCEPTION_HANDLER
);
```

The constructor signatures are similar to the standard ANSI string constructors, except for the additional `encoding` argument. A null `encoding` argument, `encoding=0`, implies the constructor uses the local character set.

16-bit character constructor

Example 133 shows the constructor that can be used to initialize an `IT_UString` with an array of 16-bit characters (represented by `unsigned short*`).

Example 133: *16-Bit Character Constructor*

```
IT_UString(
    const unsigned short* sb,
    const IT_String&      encoding,
    size_t                n = npos,
    IT_ExceptionHandler&  eh = IT_EXCEPTION_HANDLER
);
```

wchar_t character constructor

Example 134 shows the constructor that can be used to initialize an `IT_UString` with an array of `wchar_t` characters.

Example 134:*wchar_t Character Constructor*

```
IT_UString(
    const wchar_t*         wb,
    size_t                 n = npos,
    IT_ExceptionHandler&   eh = IT_EXCEPTION_HANDLER
);
```

String conversion functions

The member functions shown in Example 135 are used to convert an `IT_Bus::String` to an ordinary C-style string, a UTF-16 format string and a `wchar_t` format string:

Example 135:*String Conversion Functions*

```
// C++
const char* c_str(
    const char* encoding = 0
) const;  // has NUL character at end

const unsigned short* utf16_str() const;

const wchar_t*        wchar_t_str() const;
```

If you want to copy the return value from a string conversion function, you also need to know the dimension of the relevant array. For this, you can use the `IT_Bus::String::length()` function:

```
// C++
size_t length() const;
```

The `IT_Bus::String::length()` function returns the number of underlying characters in a string, irrespective of how many bytes it takes to represent each character. Hence, the size of the array required to hold a copy of a converted string equals `length()+1` (an extra array element is required for the `NUL` character).

String conversion examples

Example 136 shows you how to convert and copy a string, `s`, into a C-style string, a UTF-16 format string and a `wchar_t` format string.

Example 136:*String Conversion Examples*

```
// C++
// Copy 's' into a plain 'char *' string:
char *s_copy = new char[s.length()+1];
strcpy(s_copy, s.c_str());
```

Example 136: *String Conversion Examples*

```
// Copy 's' into a UTF-16 string:
unsigned short* utf16_copy = new unsigned short[s.length()+1];
const unsigned short* utf16_p = s.utf16_str();
for (i=0; i<s.length()+1; i++) {
    utf16_copy[i] = utf16_p[i];
}

// Copy 's' into a wchar_t string:
wchar_t* wchar_t_copy = new wchar_t[s.length()+1];
const wchar_t* wchar_t_p = s.wchar_t_str();
for (i=0; i<s.length()+1; i++) {
    wchar_t_copy[i] = wchar_t_p[i];
}
```

Reference

For more details about C++ ANSI strings, see ***The C++ Programming Language***, third edition, by Bjarne Stroustrup.

For more details about internationalization in Artix, see the "*Using Artix with International Codesets*" chapter of the ***Configuring and Deploying Artix Solutions*** document.

NormalizedString and Token Types

This subsection describes the syntax and C++ mapping for the `xsd:normalizedString` type, the `xsd:token` type, and all of the types deriving from `xsd:token`.

normalizedString type

A *normalized string* is a string that does not contain the return (`0x0D`), line feed (`0x0A`) or tab (`0x09`) characters. Spaces (`0x20`) are allowed, however.

token types

The token type and the types derived from token are described in Table 33.

Table 33: *Description of token and Types Derived from token*

XML Schema Type	Sample Value	Description of Value
`xsd:token`	`Only single spaces; no leading or trailing!`	Like an `xsd:normalizedString` type, except that there can be no sequences of two or more spaces (`0x20`) and no leading or trailing spaces.
`xsd:language`	`en-US`	Any language identification tag as specified in RFC 3066 (http://www.ietf.org/rfc/rfc3066.txt).

Table 33: *Description of token and Types Derived from token*

XML Schema Type	Sample Value	Description of Value
xsd:NMTOKEN	NoSpacesAllowed	Like an xsd:token type, except that spaces (0x20) are disallowed (see "Formal definitions" on page 280).
xsd:NMTOKENS	Tok01 Tok02 Tok03	A list of xsd:NMTOKEN items, using the space character as a delimiter.
xsd:Name	RestrictFirstChar	Like an xsd:token type, except that the first character is restricted to be one of Letter, '_', or ':' (see "Formal definitions" on page 280).
xsd:NCName	NoColonsAllowed	Like an xsd:Name type, except that colons, ':', are disallowed (a *non-colonized name*). See "Formal definitions" on page 280. This type is useful for constructing identifiers that use the colon, ':', as a delimiter. For example, the NCName type is used both for the prefix and the local part of an xsd:QName.
xsd:ID	LikeNCName	Like an xsd:NCName type. The xsd:ID type is a legacy from early XML specifications, where it can provide a unique ID for an XML element. The element can then be cross-referenced using the ID value.

Formal definitions

The Name, NCName, NMTOKEN, and NMTOKENS types are formally defined as follows:

```
[1]    NameChar    ::=    Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar | Extender
[2]    Name        ::=    (Letter | '_' | ':') ( NameChar)*
[3]    Names       ::=    Name (#x20 Name)*

[4]    NMTOKEN     ::=    (NameChar)+
[5]    NMTOKENS    ::=    NMTOKEN (#x20 NMTOKEN)*

[6]    NCNameChar  ::=  Letter | Digit | '.' | '-' | '_' | CombiningChar | Extender
[7]    NCName      ::=  (Letter | '_') (NCNameChar)*
```

The Name, NMTOKEN, and NMTOKENS types are defined in the *Extensible Markup Language (XML) 1.0 (Second Edition)* document (http://www.w3.org/TR/2000/WD-xml-2e-20000814). The NCName type is defined in the *Namespaces in XML* document (http://www.w3.org/TR/1999/REC-xml-names-19990114/).

The terms, CombiningChar and Extender, are defined in the *Unicode Character Database* (http://www.unicode.org/Public/UNIDATA/UCD.html). A *combining character* is a character that combines with a preceding base character—for example, accents, diacritics, Hebrew points, Arab vowel signs and Indic matras. An *extender* is a character that extends the value or shape of a preceding alphabetic character—for example, the Catalan middle dot.

C++ mapping for all token types except xsd:NMTOKENS

The token type and its derived types map to C++ as shown in Table 31 on page 273. All of the token types, except for `IT_Bus::NMTokens`, provide two constructors:

- A no-argument constructor, and
- A constructor that takes a `const IT_Bus::String&` argument.

For setting and getting a token value, the following functions are provided (inherited from `IT_Bus::NormalizedString`):

```
// C++
const String&
get_value() const IT_THROW_DECL(());

void
set_value(const String& value)
   IT_THROW_DECL((IT_Bus::Exception));
```

Validity testing functions

In addition to the functions inherited from `IT_Bus::NormalizedString`, each of the derived token types has a validity testing function, as shown in Table 34.

Table 34: *Validity Testing Functions for Normalized Strings and Tokens*

XML Schema Type	Validity Testing Function

`xsd:normalizedString`	`static bool`
	`IT_Bus::NormalizedString::is_valid_normalized_string(`
	`    const String& value`
	`)`

`xsd:token`	`static bool` `IT_Bus::Token::is_valid_token(const String& value)`
`xsd:language`	`static bool` `IT_Bus::Language::is_valid_language(const String& value)`
`xsd:NMTOKEN`	`static bool` `IT_Bus::NMToken::is_valid_nmtoken(const String& value)`
`xsd:Name`	`static bool` `IT_Bus::Name::is_valid_name(const String& value)`
`xsd:NCName`	`static bool` `IT_Bus::NCName:is_valid_ncname(const String& value)`
`xsd:ID`	`static bool` `IT_Bus::ID::is_valid_id(const String& value)`

C++ mapping of NMTOKENS

The `xsd:NMTOKENS` type maps to the C++ class, `IT_Bus::NMTokens`. The `IT_Bus::NMTokens` class inherits from `SimpleTypesListT<IT_Bus::NMToken>`, which in turn inherits from `IT_Vector<IT_Bus::NMToken>`.

The `IT_Bus::NMTokens` type is thus effectively a vector, where the element type is `IT_Bus::NMToken`. You can use the indexing operator, `[]`, to access individual elements and, in addition, the `SimpleTypesList` base class provides `set_size()` and `get_size()` functions.

For more details about `IT_Vector<T>` types, see "IT_Vector Template Class" on page 408.

Example

The following example shows how to initialize an `xsd:token` instance in C++.

```
// C++

// Test and set an xsd:token value.
IT_Bus::String tok_string = "0123 A token with spaces";
IT_Bus::Token tok;

if (IT_Bus::Token::is_valid_token(tok_string)) {
    tok.set_value(tok_string);
}
```

QName Type

`xsd:QName` maps to `IT_Bus::QName`. A qualified name, or QName, is the unique name of a tag appearing in an XML document, consisting of a *namespace URI* and a *local part*.

QName constructor

The usual way to construct an `IT_Bus::QName` object is by calling the following constructor:

```
// C++
QName::QName(
  const String &    namespace_prefix,
  const String &    local_part,
  const String &    namespace_uri
)
```

Because the namespace prefix is relatively unimportant, you can leave it blank. For example, to create a QName for the `soap:address` element:

```
// C++
IT_Bus::QName soap_address = new IT_Bus::QName(
    "",
    "address",
    "http://schemas.xmlsoap.org/wsdl/soap"
);
```

QName member functions

The `IT_Bus::QName` class has the following public member functions:

```
const IT_Bus::String &
get_namespace_prefix() const;

const IT_Bus::String &
get_local_part() const;

const IT_Bus::String &
get_namespace_uri() const;

const IT_Bus::String get_raw_name() const;
const IT_Bus::String to_string() const;
bool has_unresolved_prefix() const;
size_t get_hash_code() const;
```

QName equality

The `==` operator can be used to test for equality of `IT_Bus::QName` objects. QNames are tested for equality as follows:

1. Assuming that a namespace URI is defined for the QNames, the QNames are equal if their namespace URIs match and the local part of their element names match.
2. If one of the QNames lacks a namespace URI (empty string), the QNames are equal if their namespace prefixes match and the local part of their element names match.

Date and Time Types

The `xsd:dateTime` maps to `IT_Bus::DateTime`, which is declared in `<it_bus/date_time.h>`. `DateTime` has the following fields:

Table 35: *Member Fields of IT_Bus::DateTime*

Field	Datatype	Accessor Methods
4 digit year	short	short getYear() void setYear(short wYear)
2 digit month	short	short getMonth() void setMonth(short wMonth)

Table 35: *Member Fields of IT_Bus::DateTime*

Field	Datatype	Accessor Methods
2 digit day	`short`	`short getDay()` `void setDay(short wDay)`
hours in military time	`short`	`short getHour()` `void setHour(short wHour)`
minutes	`short`	`short getMinute()` `void setMinute(short wMinute)`
seconds	`short`	`short getSecond()` `void setSecond(short wSecond)`
milliseconds	`short`	`short getMilliseconds()` `void setMilliseconds(short wMilliseconds)`
local time zone flag		`void setLocalTimeZone()` `bool haveUTCTimeZoneOffset() const`
hour offset from GMT	`short`	`void setUTCTimeZoneOffset( short hour_offset, short minute_offset)` `void getUTCTimeZoneOffset( short & hour_offset, short & minute_offset)`
minute offset from GMT	`short`	

IT_Bus::DateTime constructor

The default constructor takes no parameters, initializing the year, month, and day fields to 1 and the other fields to 0. An alternative constructor is provided, which accepts all of the individual date/time fields, as follows:

```
IT_DateTime(short wYear, short wMonth, short wDay,
            short wHour = 0, short wMinute = 0,
            short wSecond = 0, short wMilliseconds = 0)
```

Other date and time types

Artix supports a variety of other date and time types, as shown in Table 36. Each of these types—for example, `xsd:time` and `xsd:day`—support a subset of the fields from `xsd:dateTime`. Table 36 shows which fields are supported for each date and time type; the accessors for each field are given by Table 35.

Table 36: *Member Fields Supported by Other Date and Time Types*

Date/Time Type	C++ Class	Supported Fields
`xsd:date`	`IT_Bus::Date`	year, month, day, local time zone flag, hour and minute offset from GMT.

Table 36: *Member Fields Supported by Other Date and Time Types*

Date/Time Type	C++ Class	Supported Fields
`xsd:time`	`IT_Bus::Time`	hours, minutes, seconds, milliseconds, local time zone flag, hour and minute offset from GMT.
`xsd:gDay`	`IT_Bus::GDay`	day, local time zone flag, hour and minute offset from GMT.
`xsd:gMonth`	`IT_Bus::GMonth`	month, local time zone flag, hour and minute offset from GMT.
`xsd:gMonthDay`	`IT_Bus::GMonthDay`	month, day, local time zone flag, hour and minute offset from GMT.
`xsd:gYear`	`IT_Bus::GYear`	year, local time zone flag, hour and minute offset from GMT.
`xsd:gYearMonth`	`IT_Bus::GYearMonth`	year, month, local time zone flag, hour and minute offset from GMT.

Duration Type

The `xsd:duration` type maps to `IT_Bus::Duration`, which is declared in <it_bus/duration.h>. 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.

Despite its practicality, the duration type is a fairly peculiar way of representing a time interval, because it is an *indeterminate* quantity. Both the number of days in a month and the number of days in a year can vary, depending on what you choose as the starting date of the duration.

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.

Unsupported facets

The following facets are unsupported by the `xsd:duration` element:

- `pattern`
- `whiteSpace`
- `maxInclusive`
- `maxExclusive`
- `minInclusive`
- `minExclusive`

Supported facets

The following facets are supported and checked at runtime:

- `enumeration`

Duration constructors

The `IT_Bus::Duration` class supports the constructors shown in Example 137.

Example 137: *IT_Bus::Duration Constructors*

```
// C++
Duration() IT_THROW_DECL(());

Duration(
    bool isNegative,
    IT_Bus::Long years,
    IT_Bus::Long months,
    IT_Bus::Long days,
    IT_Bus::Long hours,
    IT_Bus::Long minutes,
    IT_Bus::Long seconds,
    IT_Bus::Long milliseconds
) IT_THROW_DECL((Exception));

Duration(
    const char* value
) IT_THROW_DECL((Exception));

Duration(
    const IT_Bus::String& value
) IT_THROW_DECL((Exception));
```

These constructors enable you to specify each of the six fields of the duration: years, months, days, hours, minutes and seconds (where the seconds field is split into two arguments, seconds and milliseconds). The last two constructors enable you to initialize the duration from a lexical string. For example, a period of 1 year, 12 hours and 30 minutes can be initialized as follows:

```
// C++
IT_Bus::Duration period("P1Y0M0DT12H30M0S");
```

In the second constructor, you can leave a particular field unset by supplying a negative integer argument. For example, to represent a duration of 1 year 6 months, with the remaining fields left unset:

```
// C++
IT_Bus::Duration year_month(false, 1, 6, -1, -1, -1, -1, -1);
```

This is equivalent to calling the string value constructor as follows:

```
// C++
IT_Bus::Duration year_month("P1Y6M");
```

Duration accessors and modifiers

The accessor and modifier functions for each of the `IT_Bus::Duration` time fields are shown in Example 37.

Table 37: *Accessors and Modifier Functions for Duration Class*

Field	Accessor / Modifier
Sign	`bool is_negative()` `void set_is_negative(bool is_negative)`
Years	`IT_Bus::Long get_years()` `void set_years(IT_Bus::Long years)`
Months	`IT_Bus::Long get_months()` `void set_months(IT_Bus::Long months)`
Days	`IT_Bus::Long get_days()` `void set_days(IT_Bus::Long days)`
Hours	`IT_Bus::Long get_hours()` `void set_hours(IT_Bus::Long hours)`
Minutes	`IT_Bus::Long get_minutes()` `void set_minutes(IT_Bus::Long minutes)`

Seconds and milliseconds	`IT_Bus::Long get_seconds()`
	`IT_Bus::Long get_seconds_fraction()`
	`void set_seconds(`
	`    IT_Bus::Long seconds,`
	`    IT_Bus::Long milliseconds`
	`)`

If you pass a negative integer to a modifier function (for example, `set_years(-1)`), the corresponding time field becomes unset. If you try to access a field that is not set (for example, `get_years()`), the accessor returns zero.

In most respects, an unset time field is equivalent to a zero value. Whether or not a field is set or unset, however, does effect string conversion. See "String conversions" on page 288.

Duration equality

The `Duration` class provides equality testing operators, `==` and `!=`. For the purposes of equality testing, any unset field is treated as zero. The comparison algorithm works as follows:

1. Compute the number of months represented by the years and months items for each duration. If the computed values are different, the durations are not equal.
2. Compute the number of milliseconds represented by the days, hours, minutes and seconds (including fractional part) items for each duration. If the computed values are different, the durations are not equal.
3. Otherwise the durations are equal.

String conversions

The following member functions are provided to convert a `Duration` object to and from a string:

```
// C++
IT_Bus::String
to_string() const IT_THROW_DECL(());

void
from_string(const String& str) IT_THROW_DECL((Exception));
```

When generating a string from a `Duration` using `to_string()`, only the fields that are actually set will generate any output. See Table 38 for some examples of durations and their corresponding strings.

Table 38: *Examples of Duration String Conversion*

Duration to Convert	Output String
`Duration(false, 1, -1, -1, 0, 0, 0, -1)`	`P1YT0H0M0S`
`Duration(false, 1, -1, -1, 0, 0, 0, 0)`	`P1YT0H0M0.000S`
`Duration(false, -1, -1, -1, -1, -1, -1, -1)`	`P0D`

The example in the last row converts to a string with a single field, `0D`, although all of the fields were specified as unset. The XML schema specification requires that at least one field must be present in a duration string.

Adding a duration to a duration

You can add and subtract durations from each other using the + and - operators.

Adding a duration to a dateTime

The algorithm for adding a duration to a dateTime value is somewhat complicated, because durations involving years and months are inherently ambiguous (for example, a year might last 365 days or 366 days; a month might last 28, 29, 30, or 31 days).

The addition algorithm adopted by the XML specification tries to be as natural as possible. For example, if you add one month, P1M, to March 31, 2006, this cannot give April 31, 2006, because there is no such date. The addition algorithm therefore changes this result to April 30, 2006.

For full details of the addition algorithm, consult the XML schema specification:

http://www.w3.org/TR/xmlschema-2/#adding-durations-to-dateTimes

Adding a duration to other time types

You can also add a duration to other time and date types:

xsd:date
xsd:time
xsd:gYearMonth
xsd:gYear
xsd:gMonthDay
xsd:gMonth
xsd:gDay

Adding a duration to one of these types is performed as follows:

1. Convert the time type to a dateTime type.
2. Add the duration to the dateTime type.
3. Convert the dateTime type back to the original time type by discarding the fields that do not belong in the original time type.

Decimal Type

xsd:decimal maps to IT_Bus::Decimal, which is implemented by the Artix foundation class IT_FixedPoint, defined in <it_dsa/fixed_point.h>. IT_FixedPoint provides full fixed point decimal calculation logic using the standard C++ operators.

Note: Although the XML schema specifies that xsd:decimal has unlimited precision, the IT_FixedPoint type can have at most 31 digit precision.

IT_Bus::Decimal operators

The `IT_Bus::Decimal` type supports a full complement of arithmetical operators. See Table 39 for a list of supported operators.

Table 39: *Operators Supported by IT_Bus::Decimal*

Description	Operators
Arithmetical operators	`+, -, *, /, ++, --`
Assignment operators	`=, +=, -=, *=, /=`
Comparison operators	`==, !=, >, <, >=, <=`

IT_Bus::Decimal member functions

The following member functions are supported by `IT_Bus::Decimal`:

```
// C++
IT_Bus::Decimal round(unsigned short scale) const;

IT_Bus::Decimal truncate(unsigned short scale) const;

unsigned short number_of_digits() const;

unsigned short scale() const;

IT_Bool is_negative() const;

int compare(const IT_FixedPoint& val) const;

IT_Bus::Decimal::DigitIterator left_most_digit() const;
IT_Bus::Decimal::DigitIterator past_right_most_digit() const;
```

IT_Bus::Decimal::DigitIterator

The `IT_Bus::Decimal::DigitIterator` type is an ANSI-style iterator class that iterates over all the digits in a fixed point decimal instance.

Example

The following C++ example shows how to perform some elementary arithmetic using the `IT_Bus::Decimal` type.

```
// C++
IT_Bus::Decimal d1 = "123.456";
IT_Bus::Decimal d2 = "87654.321";

IT_Bus::Decimal d3 = d1+d2;
d3 *= d1;
if (d3 > 100000) {
   cout << "d3 = " << d3;
}
```

Integer Types

The XML schema defines the following unlimited precision integer types, as shown in Table 40.

Table 40: *Unlimited Precision Integer Types*

XML Schema Type	C++ Type
xsd:integer	IT_Bus::Integer
xsd:positiveInteger	IT_Bus::PositiveInteger
xsd:negativeInteger	IT_Bus::NegativeInteger
xsd:nonPositiveInteger	IT_Bus::NonPositiveInteger
xsd:nonNegativeInteger	IT_Bus::NonNegativeInteger

In C++, `IT_Bus::Integer` serves as the base class for `IT_Bus::PositiveInteger`, `IT_Bus::NegativeInteger`, `IT_Bus::NonPositiveInteger`, and `IT_Bus::NegativeInteger`. The lexical representation of an integer is a decimal integer with optional sign (`+` or `-`) and optional leading zeroes.

Maximum precision

In practice the precision of the integer types in Artix is not unlimited, because their internal representation uses `IT_FixedPoint`, which is limited to 31-digits.

Integer operators

The integer types supports a full complement of arithmetical operators. See Table 41 for a list of supported operators.

Table 41: *Operators Supported by the Integer Types*

Description	Operators
Arithmetical operators	`+`, `-`, `*`, `/`, `++`, `--`
Assignment operators	`=`, `+=`, `-=`, `*=`, `/=`
Comparison operators	`==`, `!=`, `>`, `<`, `>=`, `<=`

Constructors

The Artix integer classes define constructors for the following built-in integer types: `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, and `decimal`.

Alternatively, you can initialize an Artix integer from a string, using either of the following string types: `char*` and `IT_Bus::String`.

Integer member functions

The following member functions are supported by the integer types:

```
// C++
// Get value as a Decimal type
const IT_Bus::Decimal& get_value() const IT_THROW_DECL(());

// Set value as a Decimal type.
// Passing a true value for the 'truncate' parameter causes the
// constructor to truncate 'value' at the decimal point.
void set_value(
    const IT_Bus::Decimal& value,
    bool truncate = false
) IT_THROW_DECL((IT_Bus::Exception));

// Return true if integer value is less than zero
IT_Bus::IT_Bool is_negative() const;

// Return true if integer value is greater than zero
IT_Bus::IT_Bool is_positive() const;

// Return true if integer value is greater than or equal to zero
IT_Bus::IT_Bool is_non_negative() const;

// Return true if integer value is less than or equal to zero
IT_Bus::IT_Bool is_non_positive() const;

// Return true if the decimal 'value' has no fractional part
static bool is_valid_integer(const IT_Bus::Decimal& value)
   const;

// Return 1, if this instance is greater than 'other'.
// Return 0, if this instance is equal to 'other'.
// Return -1, if this instance is smaller than 'other'.
int compare(const Integer& other) const;

// Convert to IT_Bus::String
const IT_Bus::String to_string() const;
```

Example

The following C++ example shows how to perform some elementary arithmetic using the `IT_Bus::Integer` type.

```
// C++
IT_Bus::Integer i1 = "321";
IT_Bus::Integer i2 = "87654";

IT_Bus::Integer i3 = i1 + i2;
i3 *= i1;
if (i3 > 100000) {
   cout << "i3 = " << i3.to_string() << endl;
}
```

Mixed arithmetic

You can mix different integer types in an arithmetic expression, but the result is always of `IT_Bus::Integer` type. For example, you could mix the `IT_Bus::PositiveInteger` and `IT_Bus::NegativeInteger` types in an arithmetic expression as follows:

```
// C++
IT_Bus::PositiveInteger p1(+100), p2(+200);
IT_Bus::NegativeInteger n1(-500);

IT_Bus::Integer = (p1 + n1) * p2;
```

Binary Types

The WSDL binary types map to C++ as shown in Table 42:

Table 42: *Schema to Bus Mapping for the Binary Types*

Schema Type	Bus Type
xsd:base64Binary	IT_Bus::Base64Binary
xsd:hexBinary	IT_Bus::HexBinary
xmime:base64Binary	IT_Bus::XMimeBase64Binary
xmime:hexBinary	IT_Bus::XMimeHexBinary

Regular encodings

The difference between `HexBinary` and `Base64Binary` is the way they are encoded for transmission. The `Base64Binary` encoding is more compact because it uses a larger set of symbols in the encoding. The encodings can be compared as follows:

- `HexBinary`—the hex encoding uses a set of 16 symbols `[0-9a-fA-F]`, ignoring case, and each character can encode 4 bits. Hence, two characters represent 1 byte (8 bits).

- Base64Binary—the base 64 encoding uses a set of 64 symbols and each character can encode 6 bits. Hence, four characters represent 3 bytes (24 bits).

XMIME encodings

The XMimeBase64Binary and XMimeHexBinary types are meant to be used in conjunction with the MTOM transmission optimization. For details, see "Binary Types and MTOM" on page 342.

IT_Bus::Base64Binary and IT_Bus::HexBinary

Both the IT_Bus::Base64Binary and the IT_Bus::HexBinary classes expose the following member functions to access the buffer value:

```
// C++
virtual const BinaryBuffer &
get_buffer() const;

virtual BinaryBuffer &
get_buffer();
```

The first form of get_buffer() returns a read-only reference to the binary buffer. The second form of get_buffer() returns a modifiable reference to the binary buffer.

IT_Bus::XMimeBase64Binary and IT_Bus::XMimeHexBinary

Both the IT_Bus::XMimeBase64Binary and the IT_Bus::XMimeHexBinary classes expose the following member functions:

```
// C++
String &
get_content_type();

const String &
get_content_type() const;

void
set_content_type(const String & val);

virtual const BinaryBuffer &
get_buffer() const;

virtual BinaryBuffer &
get_buffer();
```

In addition to the buffer accessors and modifiers, these types provide functions to access and modify the data's MIME content type.

IT_Bus::BinaryBuffer class

You can perform buffer manipulation by invoking the member functions of the `IT_Bus::BinaryBuffer` class. A binary buffer instance is a contiguous data buffer that encapsulates the following information:

- *Null-terminated string*—internally, a binary buffer is represented as a null-terminated string (C style string). The terminating `NULL` character is not counted in the buffer size.
- *Borrowing flag*—internally, the binary buffer keeps track of whether it *owns* the buffer memory (in which case the binary buffer is responsible for deleting it) or whether the binary buffer merely *borrows* the buffer memory (in which case the binary buffer is not responsible for deleting it).

Allocating and deallocating binary buffers

Example 138 shows the signatures of the binary buffer functions for allocating and deallocating binary buffers.

Example 138: *Functions for Allocating and Deallocating Binary Buffers*

```
// C++
BinaryBuffer()

BinaryBuffer(IT_Bus::String rhs);

BinaryBuffer(const char * data, long size = -1);

virtual ~BinaryBuffer();

void allocate(long size);

void resize(long size);

void clear();
```

The preceding binary buffer functions can be described as follows:

- `BinaryBuffer` constructors—you can construct a binary buffer either by passing in an `IT_Bus::String` instance or a pointer to a `const char *`. In both cases, the binary buffer makes its own copy of the data.
- `BinaryBuffer` destructor—if the borrowing flag is false, the destructor deletes the memory for the buffer data.
- `allocate()` function—allocates a new buffer of the specified size.
- `resize()` function—an optimized allocation function that attempts to reuse the existing buffer, if possible. This function throws an `IT_Bus::Exception`, if it is called on a borrowed buffer.
- `clear()` function—resets the binary buffer to an empty buffer. If the buffer data is not borrowed, it deletes the old memory.

Assigning and copying binary buffers

Example 139 shows the signatures of the binary buffer functions for assigning and copying binary buffers.

Example 139:*Functions for Assigning and Copying Binary Buffers*

```
// C++
// Copying assignments
void operator=(const BinaryBuffer& rhs);
void operator=(IT_Bus::String rhs);
void operator=(const char* rhs);

BinaryBuffer& assign(const String & rhs, size_t n);
BinaryBuffer& assign(const char* rhs, size_t n);

void copy(const char* p, long size = -1);

// Non-copying assignments
void attach(BinaryBuffer& attach_buffer);

void attach_external(char* p, long size, bool borrow = true);

void borrow(const BinaryBuffer& borrow_buffer);
void borrow(const char* borrow_data, long size = -1);
```

The copying assignment functions can be described as follows:

- `operator=()` operator—you can assign another `BinaryBuffer` instance, an `IT_Bus::String` instance, or a `const char *` string to a binary buffer using `operator=()`. In each of these cases, the binary buffer makes its own copy of the data and sets the borrowing flag to `false`.
- `assign()` function—similar to `operator=()`, except that you can specify the size of the string to copy. If the specified size, `n`, is less than the actual size of the string, the copied string is truncated to include only the first `n` characters.
- `copy()` function—the same as the `assign()` function, except that `copy()` returns the `void` type, instead of `BinaryBuffer&`.

The non-copying assignment functions can be described as follows:

- `attach()` function—sets this binary buffer's data pointer to point at the data in the `attach_buffer` binary buffer, taking ownership of the data if possible (in other words, this binary buffer's borrowing flag is set equal to the `attach_buffer`'s borrowing flag). The `attach_buffer` binary buffer is cleared.
- `attach_external()` function—sets the binary buffer's data pointer equal to the `char *` argument, `p`, but does *not* attempt to take ownership of the data by default. However, if you explicitly specify the `borrow` argument to be false, the binary buffer does take ownership of the data.
- `borrow()` function—sets this binary buffer's data pointer to point at the data in the `borrow_buffer` binary buffer (or `borrow_data` string, as the case may be), but does *not* take ownership of the data (in other words, this binary buffer's borrowing flag is set to `true` in all cases).

Accessing binary buffer data

Example 140 shows the signatures of the binary buffer functions for accessing binary buffer data.

Example 140:*Functions for Accessing Binary Buffer Data*

```
// C++
char operator[](long lIndex);

char* at(long lIndex);

char* get_pointer();

const char* get_const_pointer() const;

long get_size() const;

IT_String get_it_string() const;

String get_string() const;
```

The preceding binary buffer functions can be described as follows:

- `operator[]()` operator—accesses the character at position `lIndex`. The index must lie in the range `[0, get_size()]`, where the last accessible character is the terminating `NULL` character. If the index is out of range, an `IT_Bus::Exception` is thrown.
- `at()` function—similar to `operator[]()`, except that a pointer to char is returned.
- `get_pointer()` function—returns a pointer to the first character of the buffer for reading and writing (equivalent to `at(0)`).
- `get_const_pointer()` function—returns a pointer to the first character of the buffer, for read-only operations.
- `get_size()` function—returns the size of the buffer (not including the terminating `NULL` character).
- `get_it_string()` function—converts the buffer data to an `IT_String` type.
- `get_string()` function—converts the buffer data to an `IT_Bus::String` type.

Searching and comparing binary buffers

Example 141 shows the signatures of the binary buffer functions for searching and comparing binary buffers.

Example 141:*Functions for Searching and Comparing Binary Buffers*

```
// C++
char* instr(char c, long lIndex = 0);

String substr(long lIndex, long size = -1) const;

long find(const char* s, long lIndex = 0) const;

long find_binary_buffer(long& dwFindIdx, long dwFindMaxIdx,
   BinaryBuffer& vvPacketTerminator) const;

bool operator==(const BinaryBuffer & rhs) const;
```

The preceding binary buffer functions can be described as follows:

- `instr()` function—returns a pointer to the first occurrence of the character, `c`, in the buffer, where the search begins at the specified index value, `lIndex`.
- `substr()` function—returns a sub-string from the buffer, starting at the index, `lIndex`, and continuing for `size` characters (the defaulted `size` value, `-1`, selects up to the end of the buffer)
- `find()` function—returns the position of the first occurrence of the string, s, inside the buffer. The `lIndex` parameter can be used to specify the point in the buffer from which the search begins.
- `find_binary_buffer()` function—returns the position of the first occurrence of the `vvPacketTerminator` buffer within the specified buffer sub-range, `[dwFindIdx, dwFindMaxIdx]`. At the end of the search, the `dwFindIdx` parameter is equal to the found position.
- `operator==()` operator—comparison is `true`, if the compared buffers are of the same length and have identical contents; otherwise, `false`.

Concatenating binary buffers

Example 142 shows the signatures of the binary buffer functions for concatenating binary buffers.

Example 142:*Functions for Concatenating Binary Buffers*

```
// C++
char* concat(const char* szThisString, long size = -1);
```

The preceding binary buffer function can be described as follows:

- `concat()` function—adds the string, `szThisString`, to the end of the buffer. You can specify the `size` parameter to limit the number of characters from `szThisString` that are concatenated (the default is to concatenate the whole string).

Example

Consider a port type that defines an echoHexBinary operation. The echoHexBinary operation takes an IT_Bus::HexBinary type as an in parameter and then echoes this value in the response. Example 143 shows how a server might implement the echoHexBinary operation.

Example 143: *C++ Implementation of an echoHexBinary Operation*

```
// C++
using namespace IT_Bus;
...
void BaseImpl::echoHexBinary(
    const IT_Bus::HexBinaryInParam & inputHexBinary,
    IT_Bus::HexBinaryOutParam& Response
)
    IT_THROW_DECL((IT_Bus::Exception))
{
    // Copy the input buffer to the output buffer.
    Response.get_buffer() = inputHexBinary.get_buffer();
}
```

Note: The IT_Bus::HexBinaryInParam and IT_Bus::HexBinaryOutParam types are both essentially equivalent to IT_Bus::HexBinary. These extra types help the compiler to distinguish between in parameters and out parameters. They are only used in operation signatures.

Likewise, the IT_Bus::Base64BinaryInParam and IT_Bus::Base64BinaryOutParam types are both essentially equivalent to IT_Bus::Base64Binary.

Deriving Simple Types by Restriction

Artix currently has limited support for the derivation of simple types by restriction. You can define a restricted simple type using any of the standard facets, but in most cases the restrictions are not checked at runtime.

Unchecked facets

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

- whiteSpace

Checked facets

The following facets are supported and checked at runtime:

- enumeration
- length
- maxLength

- minLength
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- pattern
- totalDigits
- fractionDigits

C++ mapping

In general, a restricted simple type, *RestrictedType*, obtained by restriction from a base type, *BaseType*, maps to a C++ class, *RestrictedType*, with the following public member functions:

```
// C++
const IT_Bus::QName & get_type() const;

void            set_value(const BaseType & value);
BaseType get_value() const;
```

Restriction with an enumeration facet

Artix supports the restriction of simple types using the enumeration facet. The base simple type can be any simple type except `xsd:boolean`.

When an enumeration type is mapped to C++, the C++ implementation of the type ensures that instances of this type can only be set to one of the enumerated values. If `set_value()` is called with an illegal value, it throws an `IT_Bus::Exception` exception.

WSDL example of enumeration facet

Example 144 shows an example of a `ColorEnum` type, which is defined by restriction from the `xsd:string` type using the enumeration facet. When defined in this way, the `ColorEnum` restricted type is only allowed to take on one of the string values `RED`, `GREEN`, or `BLUE`.

Example 144: *WSDL Example of Derivation with the Enumeration Facet*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >
    <types>
        <schema ... >
            <simpleType name="ColorEnum">
                <restriction base="xsd:string">
                    <enumeration value="RED"/>
                    <enumeration value="GREEN"/>
                    <enumeration value="BLUE"/>
                </restriction>
            </simpleType>
            ...
</definitions>
```

C++ mapping of enumeration facet

The WSDL-to-C++ compiler maps the `ColorEnum` restricted type to the `ColorEnum` C++ class, as shown in Example 145. The only values that can legally be set using the `set_value()` member function are the strings `RED`, `GREEN`, or `BLUE`.

Example 145: *C++ Mapping of ColorEnum Restricted Type*

```
// C++
class ColorEnum : public IT_Bus::AnySimpleType
{
    ...
  public:
    ColorEnum();
    ColorEnum(const IT_Bus::String & value);
    ...

    ColorEnum& operator= (const ColorEnum& assign);
    IT_Bus::Boolean operator== (const ColorEnum& copy);

    virtual const IT_Bus::QName & get_type() const;
    void           set_value(const IT_Bus::String & value);
    IT_Bus::String get_value() const;
};
```

List Type

The `xsd:list` schema type is a simple type that enables you to define space-separated lists. For example, if the `numberList` element is defined to be a list of floating point numbers, an instance of a `numberList` element could look like the following:

```
<numberList>1.234 2.345 5.432 1001</numberList>
```

XML schema supports two distinct ways of defining a list type, as follows:

- Defining list types with the itemType attribute.
- Defining list types by derivation.

Defining list types with the itemType attribute

The first way to define a list type is by specifying the list item type using the `itemType` attribute. For example, you could define the list type, `StringListType`, as a list of `xsd:string` items, with the following syntax:

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

<element name="stringList" type="StringListType"/>
```

An instance of a `stringList` element, which is defined to be of `StringListType` type, could look like the following:

```
<stringList>wool cotton linen</stringList>
```

Defining list types by derivation

The second way to define a list type is to use simple derivation. For example, you could define the list type, `IntListType`, as a list of `xsd:int` items, with the following syntax:

```
<simpleType name="IntListType">
    <list>
        <simpleType>
            <restriction base="xsd:int"/>
        </simpleType>
    </list>
</simpleType>

<element name="intList" type="IntListType"/>
```

An instance of an `intList` element, which is defined to be of `IntListType` type, could look like the following:

```
<intList>1 2 3 5 8 13 21 34 55</intList>
```

C++ mapping

In C++, lists are represented by an `IT_Vector<T>` template type. Hence, C++ list classes support the `operator[]`, to access individual items, and the `get_size()` function, to get the length of the list.

For example, the `StringListType` type defined previously would map to the `StringListType` C++ class, which inherits from `IT_Vector<IT_Bus::String>`.

Example

Given an instance of `StringListType` type, you could print out its contents as follows:

```
// C++
StringListType s_list = ... // Initialize list

for (int i=0; i < s_list.get_size(); i++)
{
    cout << s_list[i] << endl;
}
```

Union Type

The `xsd:union` schema type enables you to define an element whose type can be any of the simple types listed in the union definition. In general, the syntax for defining a union, *UnionType*, is as follows:

```
<simpleType name="UnionType">
    <union memberTypes="Type01 Type02 ...">
        <simpleType> ... </simpleType>
        <simpleType> ... </simpleType>
        ...
    </union>
</simpleType>
```

Where *Type01*, *Type02*, and so on are the names of simple types that the union could contain. The `simpleType` elements appearing within the `union` element define anonymous simple types (defined by derivation) that the union could contain.

XML schema supports the following ways of defining a union type:

- Defining union types with the memberTypes attribute.
- Defining union types by derivation.

Defining union types with the memberTypes attribute

The first way to define a union type is by specifying the list of allowable member types using the `memberTypes` attribute. For example, you could define a `UnionOfIntAndFloat` union type to contain either an `xsd:int` or an `xsd:float`, as follows:

```
<xsd:simpleType name="UnionOfIntAndFloat">
    <xsd:union memberTypes="xsd:int xsd:float"/>
</xsd:simpleType>

<xsd:element name="u1" type="UnionOfIntAndFloat"/>
```

Some sample instances of the u2 element could look like the following:

```
<u1>500</u1>
<u1>1.234e06</u1>
```

Defining union types by derivation

The second way to define a union type is by adding one or more anonymous simpleType elements to the union body. For example, you could define the UnionByDerivation type to contain either a member derived from a xsd:string or a member derived from an xsd:int, as follows:

```
<xsd:simpleType name="UnionByDerivation">
    <xsd:union>
        <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                <enumeration value="Bill"/>
                <enumeration value="Ben"/>
            </xsd:restriction>
        </xsd:simpleType>
        <xsd:simpleType>
            <xsd:restriction base="xsd:int">
                <maxInclusive value="1000"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:union>
</xsd:simpleType>

<xsd:element name="u2" type="UnionByDerivation"/>
```

Some sample instances of the u2 element could look like the following:

```
<u2>Bill</u2>
<u2>999</u2>
```

WSDL example

Example 146 shows an example of a union type, Union2, which can contain either a Union1 type or an enumerated string.

Example 146:*Definition of a Union Type in WSDL*

```
// C++
<xsd:simpleType name="Union1">
    <xsd:union memberTypes="xsd:int xsd:float"/>
</xsd:simpleType>

<xsd:simpleType name="Union2">
    <xsd:union memberTypes="tns:Union1">
        <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                <enumeration value="Tweedledum"/>
                <enumeration value="Tweedledee"/>
```

Example 146:*Definition of a Union Type in WSDL*

```
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:union>
</xsd:simpleType>
```

C++ mapping

The WSDL-to-C++ compiler maps the preceding WSDL (Example 146 on page 304) to the Union2 C++ class. An outline of this class is shown in Example 147.

Example 147:*Mapping of Union2 to C++*

```
// C++
class Union2 : public IT_Bus::SimpleTypeUnion
{
  public:

    Union2();
    Union2(const Union2 & copy);
    virtual ~Union2();

    // ...

    virtual const IT_Bus::QName & get_type() const;

    Union2 & operator=(const Union2 & rhs);

    IT_Bus::Boolean
    operator==(const Union2 & rhs) const IT_THROW_DECL(())

    IT_Bus::Boolean
    operator!=(const Union2 & rhs) const IT_THROW_DECL(());

    enum Union2Discriminator
    {
        var_Union1_enum,
        var_string_enum,
        Union2_MAXLONG=-1
    } m_discriminator;

    Union2Discriminator
    get_discriminator() const IT_THROW_DECL(())
    {
        return m_discriminator;
    }

    IT_Bus::UInt
    get_discriminator_as_uint() const IT_THROW_DECL(())
    {
        return m_discriminator;
    }

    Union1 &       getUnion1();
    const Union1 & getUnion1() const;
    void           setUnion1(const Union1 & val);
```

Example 147:*Mapping of Union2 to C++*

```
    Union2String &       getstring();
    const Union2String & getstring() const;
    void                 setstring(const Union2String & val);
    // ...
};
```

The C++ mapping defines a pair of accessor and modifier functions, get*MemberType*() and set*MemberType*(), for each union member type, *MemberType*. The name of the accessor and modifier functions are determined as follows:

- If the union member is an atomic type (for example, int or string), the functions are defined as get*AtomicType*() and set*AtomicType*() (for example, getint() and setint()).
- If the union member is a user-defined type, *UserType*, the functions are defined as get*UserType*() and set*UserType*().
- If the union member is defined by derivation (that is, using a simpleType element in the scope of the <union> tag), the accessor and modifier functions are named after the base type, *BaseType*, to yield get*BaseType*() and set*BaseType*().

Example

Consider a port type that defines an echoUnion operation. The echoUnion operation takes a Union2 type as an in parameter and then echoes this value in the response. Example 148 shows how a client could use a proxy instance, bc, to invoke the echoUnion operation.

Example 148:*Printing a Union2 Type Returned from an Operation*

```
// C++
Union2 uIn, uOut;

// Initialize uIn with the value "Tweedledum"
uIn.setstring("Tweedledum");

try {
    bc.echoUnion(uIn, uOut);

    switch (uOut.get_discriminator()) {
        case Union2::var_Union1_enum :
            switch (uOut.getUnion1().get_discriminator()) {
                case Union1::var_int_enum :
                    cout << "Result = (int) "
                         << uOut.getUnion1().getint() << endl;
                case Union1::var_float_enum :
                    cout << "Result = (float) "
                        << uOut.getUnion1().getfloat() << endl;
                    break;
            }
            break;
        case Union2::var_string_enum :
            cout << "Result = (string) "
```

Example 148: *Printing a Union2 Type Returned from an Operation*

```
                << uOut.getstring().get_value().c_str() << endl;
            break;
    }
} catch (IT_Bus::FaultException &ex)
{
    // Handle exception (not shown) ...
}
```

Holder Types

There are some general-purpose functions in Artix (for example, some functions in the context API) that take parameters of `IT_Bus::AnyType` type, which allows you to pass *any* Artix data type. You can pass most Artix data types directly to such functions, because the data types derive from the `AnyType` class. However, not all Artix data types derive from `AnyType`. Some types, such as `IT_Bus::Int` and `IT_Bus::Short`, are simply typedefs of C++ built-in types. Other simple types—for example, `IT_Bus::String` and `IT_Bus::QName`—also do not inherit from `AnyType`.

To facilitate the passing of simple types, Artix defines `Holder` types. For example, the `IT_Bus::StringHolder` type can hold an `IT_Bus::String` instance. In contrast to the original *`Simple`* type, the *`Simple`*`Holder` type derives from `IT_Bus::AnyType`. Accessor and modifier functions are used to insert and extract the *`Simple`* type from the *`Simple`*`Holder` type.

Holder type member functions

A holder type, for data of type *`T`*, supports the following accessor and modifier member functions:

```
// C++
const T& get() const;

T&       get();

void     set(const T& data);
```

Example

The following example shows how to use the `IT_Bus::StringHolder` type to set the `HTTP_ENDPOINT_URL` context value.

```
// C++
IT_Bus::AnyType* any_string = request_contexts->get_context(
    IT_ContextAttributes::HTTP_ENDPOINT_URL,
    true
);

IT_Bus::StringHolder* str_holder =
   dynamic_cast<IT_Bus::StringHolder*>(any_string);

str_holder->set("http://localhost:1234");
```

List of holder types

Table 43 shows the list of `Holder` types provided by Artix.

Table 43: *List of Artix Holder Types*

Built-In Type	Holder Type
IT_Bus::Boolean	IT_Bus::BooleanHolder
IT_Bus::Byte	IT_Bus::ByteHolder
IT_Bus::Short	IT_Bus::ShortHolder
IT_Bus::Int	IT_Bus::IntHolder
IT_Bus::Long	IT_Bus::LongHolder
IT_Bus::String	IT_Bus::StringHolder
IT_Bus::Float	IT_Bus::FloatHolder
IT_Bus::Double	IT_Bus::DoubleHolder
IT_Bus::UByte	IT_Bus::UByteHolder
IT_Bus::UShort	IT_Bus::UShortHolder
IT_Bus::UInt	IT_Bus::UIntHolder
IT_Bus::ULong	IT_Bus::ULongHolder
IT_Bus::Decimal	IT_Bus::DecimalHolder
IT_Bus::QName	IT_Bus::QNameHolder
IT_Bus::DateTime	IT_Bus::DateTimeHolder
IT_Bus::HexBinary	IT_Bus::HexBinaryHolder
IT_Bus::Base64Binary	IT_Bus::Base64BinaryHolder

Unsupported Simple Types

List of unsupported simple types

The following WSDL simple types are currently not supported by the WSDL-to-C++ compiler:

Atomic Simple Types

xsd:ENTITY
xsd:ENTITIES
xsd:NOTATION
xsd:IDREF
xsd:IDREFS

Complex Types

This section describes the WSDL-to-C++ mapping for complex types. Complex types are defined within an XML schema. In contrast to simple types, complex types can contain elements and carry attributes.

This section contains the following areas:

- Sequence Complex Types
- Choice Complex Types
- All Complex Types
- Attributes
- Attribute Groups
- Nesting Complex Types
- Deriving a Complex Type from a Simple Type
- Deriving a Complex Type from a Complex Type
- Arrays
- Model Group Definitions

Sequence Complex Types

XML schema sequence complex types are mapped to a generated C++ class, which inherits from `IT_Bus::SequenceComplexType`. The mapped C++ class is defined in the generated *PortTypeName*`Types.h` and *PortTypeName*`Types.cxx` files.

The WSDL-to-C++ mapping defines accessor and modifier functions for each element in the sequence complex type.

Occurrence constraints

Occurrence constraints, which are specified using the `minOccurs` and `maxOccurs` attributes, are supported for sequence complex types. See "Sequence Occurrence Constraints" on page 369.

WSDL example

Example 149 shows an example of a sequence, `SequenceType`, with three elements.

Example 149:*Definition of a Sequence Complex Type in WSDL*

```
<schema targetNamespace="http://soapinterop.org/xsd"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <complexType name="SequenceType">
        <sequence>
            <element name="varFloat" type="xsd:float"/>
            <element name="varInt" type="xsd:int"/>
            <element name="varString" type="xsd:string"/>
        </sequence>
    </complexType>
    ...
</schema>
```

C++ mapping

The WSDL-to-C++ compiler maps the preceding WSDL (Example 149) to the `SequenceType` C++ class. An outline of this class is shown in Example 150.

Example 150:*Mapping of SequenceType to C++*

```
// C++
class SequenceType : public IT_Bus::SequenceComplexType
{
  public:
    SequenceType();
    SequenceType(const SequenceType& copy);
    virtual ~SequenceType();
    ...
    virtual const IT_Bus::QName & get_type() const;

    SequenceType& operator= (const SequenceType& assign);

    const IT_Bus::Float & getvarFloat() const;
    IT_Bus::Float &       getvarFloat();
    void                  setvarFloat(const IT_Bus::Float & val);

    const IT_Bus::Int &   getvarInt() const;
    IT_Bus::Int &         getvarInt();
    void                  setvarInt(const IT_Bus::Int & val);

    const IT_Bus::String & getvarString() const;
    IT_Bus::String &      getvarString();
    void                  setvarString(const IT_Bus::String & val);

  private:
    ...
};
```

Each *ElementName* element declared in the sequence complex type is mapped to a pair of accessor/modifier functions, `get`*ElementName*`()` and `set`*ElementName*`()`.

Example

Consider a port type that defines an `echoSequence` operation. The `echoSequence` operation takes a `SequenceType` type as an in parameter and then echoes this value in the response. Example 151 shows how a client could use a proxy instance, `bc`, to invoke the `echoSequence` operation.

Example 151:*Client Invoking an echoSequence Operation*

```
// C++
SequenceType seqIn, seqResult;
seqIn.setvarFloat(3.14159);
seqIn.setvarInt(54321);
seqIn.setvarString("You can use a string constant here.");

try {
    bc.echoSequence(seqIn, seqResult);

    if((seqResult.getvarInt() != seqIn.getvarInt()) ||
       (seqResult.getvarFloat() != seqIn.getvarFloat()) ||
       (seqResult.getvarString().compare(seqIn.getvarString()) != 0))
    {
        cout << endl << "echoSequence FAILED" << endl;
        return;
    }
} catch (IT_Bus::FaultException &ex)
{
    cout << "Caught Unexpected FaultException" << endl;
    cout << ex.get_description().c_str() << endl;
}
```

Choice Complex Types

XML schema choice complex types are mapped to a generated C++ class, which inherits from `IT_Bus::ChoiceComplexType`. The mapped C++ class is defined in the generated *PortTypeName*`Types.h` and *PortTypeName*`Types.cxx` files.

The WSDL-to-C++ mapping defines accessor and modifier functions for each element in the choice complex type. The choice complex type is effectively equivalent to a C++ union, so only one of the elements is accessible at a time. The C++ implementation defines a discriminator, which tells you which of the elements is currently selected.

Occurrence constraints

Occurrence constraints, which are specified using the `minOccurs` and `maxOccurs` attributes, are supported for choice complex types. See "Choice Occurrence Constraints" on page 372.

WSDL example

Example 152 shows an example of a choice complex type, `ChoiceType`, with three elements.

Example 152: *Definition of a Choice Complex Type in WSDL*

```
<schema targetNamespace="http://soapinterop.org/xsd"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <complexType name="ChoiceType">
        <choice>
            <element name="varFloat" type="xsd:float"/>
            <element name="varInt" type="xsd:int"/>
            <element name="varString" type="xsd:string"/>
        </choice>
    </complexType>

    ...
</schema>
```

C++ mapping

The WSDL-to-C++ compiler maps the preceding WSDL (Example 152) to the `SequenceType` C++ class. An outline of this class is shown in Example 153.

Example 153: *Mapping of ChoiceType to C++*

```
// C++
class ChoiceType : public IT_Bus::ChoiceComplexType
{
  public:
    ChoiceType();
    ChoiceType(const ChoiceType& copy);
    virtual ~ChoiceType();

    ...
    virtual const IT_Bus::QName & get_type() const ;

    ChoiceType& operator= (const ChoiceType& assign);

    const IT_Bus::Float getvarFloat() const;
    void setvarFloat(const IT_Bus::Float& val);

    const IT_Bus::Int getvarInt() const;
    void setvarInt(const IT_Bus::Int& val);

    const IT_Bus::String& getvarString() const;
    void setvarString(const IT_Bus::String& val);
```

Example 153: *Mapping of ChoiceType to C++*

```
    ChoiceTypeDiscriminator get_discriminator() const
    {
        return m_discriminator;
    }

    IT_Bus::UInt get_discriminator_as_uint() const
    {
        return m_discriminator;
    }

    enum ChoiceTypeDiscriminator
    {
        varFloat_enum,
        varInt_enum,
        varString_enum,
        ChoiceType_MAXLONG=-1L
    } m_discriminator;

  private:
    ...
};
```

Each *ElementName* element declared in the sequence complex type is mapped to a pair of accessor/modifier functions, `get`*ElementName*`()` and `set`*ElementName*`()`.

The member functions have the following effects:

- `set`*ElementName*`()`—select the *ElementName* element, setting the discriminator to the *ElementName* label and initializing the value of *ElementName*.
- `get`*ElementName*`()`—get the value of the *ElementName* element. You should always check the discriminator before calling the `get`*ElementName*`()` accessor. If *ElementName* is not currently selected, the value returned by `get`*ElementName*`()` is undefined.
- `get_discriminator()`—returns the value of the discriminator.

Example

Consider a port type that defines an `echoChoice` operation. The `echoChoice` operation takes a `ChoiceType` type as an in parameter and then echoes this value in the response. Example 154 shows how a client could use a proxy instance, `bc`, to invoke the `echoChoice` operation.

Example 154: *Client Invoking an echoChoice Operation*

```
// C++
ChoiceType cIn, cResult;
// Initialize and select the ChoiceType::varString label.
cIn.setvarString("You can use a string constant here.");

try {
    bc.echoChoice(cIn, cResult);
```

Example 154: *Client Invoking an echoChoice Operation*

```
    bool fail = IT_TRUE;
    if (cIn.get_discriminator()==cResult.get_discriminator()) {
        switch (cIn.get_discriminator()) {
            case ChoiceType::varFloat_enum:
                fail
   =(cIn.getvarFloat()!=cResult.getvarFloat());
                break;
            case ChoiceType::varInt_enum:
                fail =(cIn.getvarInt()!=cResult.getvarInt());
                break;
            case ChoiceType::varString_enum:
                fail =
                 (cIn.getvarString()!=cResult.getvarString());
                break;
        }
    }

    if (fail) {
        cout << endl << "echoChoice FAILED" << endl;
        return;
    }
} catch (IT_Bus::FaultException &ex)
{
    cout << "Caught Unexpected FaultException" << endl;
    cout << ex.get_description().c_str() << endl;
}
```

All Complex Types

XML schema all complex types are mapped to a generated C++ class, which inherits from `IT_Bus::AllComplexType`. The mapped C++ class is defined in the generated *PortTypeName*`Types.h` and *PortTypeName*`Types.cxx` files.

The WSDL-to-C++ mapping defines accessor and modifier functions for each element in the all complex type. With an all complex type, the order in which the elements are transmitted is immaterial.

Note: An all complex type can only be declared as the *outermost* group of a complex type. Hence, you cannot nest an `all` model group, `<all>`, directly inside other model groups, `<all>`, `<sequence>`, or `<choice>`. You may, however, define an `all` complex type and then declare an element of that type within the scope of another model group.

Occurrence constraints

Occurrence constraints are supported for the elements of XML schema all complex types.

WSDL example

Example 155 shows an example of an all complex type, `AllType`, with three elements.

Example 155:*Definition of an All Complex Type in WSDL*

```
<schema targetNamespace="http://soapinterop.org/xsd"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <complexType name="AllType">
        <all>
            <element name="varFloat" type="xsd:float"/>
            <element name="varInt" type="xsd:int"/>
            <element name="varString" type="xsd:string"/>
        </all>
    </complexType>
    ...
</schema>
```

C++ mapping

The WSDL-to-C++ compiler maps the preceding WSDL (Example 155) to the `AllType` C++ class. An outline of this class is shown in Example 156.

Example 156:*Mapping of AllType to C++*

```
// C++
class AllType : public IT_Bus::AllComplexType
{
  public:
    AllType();
    AllType(const AllType& copy);
    virtual ~AllType();

    virtual const IT_Bus::QName & get_type() const;

    AllType& operator= (const AllType& assign);

    const IT_Bus::Float & getvarFloat() const;
    IT_Bus::Float & getvarFloat();
    void setvarFloat(const IT_Bus::Float & val);

    const IT_Bus::Int & getvarInt() const;
    IT_Bus::Int & getvarInt();
    void setvarInt(const IT_Bus::Int & val);

    const IT_Bus::String & getvarString() const;
    IT_Bus::String & getvarString();
    void setvarString(const IT_Bus::String & val);

  private:
    ...
};
```

Each *ElementName* element declared in the sequence complex type is mapped to a pair of accessor/modifier functions, get*ElementName*() and set*ElementName*().

Example

Consider a port type that defines an echoAll operation. The echoAll operation takes an AllType type as an in parameter and then echoes this value in the response. Example 157 shows how a client could use a proxy instance, bc, to invoke the echoAll operation.

Example 157:*Client Invoking an echoAll Operation*

```
// C++
AllType allIn, allResult;
allIn.setvarFloat(3.14159);
allIn.setvarInt(54321);
allIn.setvarString("You can use a string constant here.");

try {
    bc.echoAll(allIn, allResult);

    if((allResult.getvarInt() != allIn.getvarInt()) ||
       (allResult.getvarFloat() != allIn.getvarFloat()) ||
       (allResult.getvarString().compare(allIn.getvarString()) != 0))
    {
        cout << endl << "echoAll FAILED" << endl;
        return;
    }
} catch (IT_Bus::FaultException &ex)
{
    cout << "Caught Unexpected FaultException" << endl;
    cout << ex.get_description().c_str() << endl;
}
```

Attributes

Artix supports the use of <attribute> declarations within the scope of a <complexType> definition. For example, you can include attributes in the definitions of an all complex type, sequence complex type, and choice complex type. The declaration of an attribute in a complex type has the following syntax:

```
<attribute name="AttrName" type="AttrType"
   use="[optional|required|prohibited]"/>
```

Attribute use

When declaring an attribute, the use can have one of the following values:

- optional—(default) the attribute can either be set or unset.
- required—the attribute must be set.
- prohibited—the attribute must be unset (cannot be used).

On-the-wire optimization

Artix optimizes the transmission of attributes by distinguishing between set and unset attributes. Only *set* attributes are transmitted (on bindings that support this optimization).

Note: The CORBA binding does not support this optimization.

C++ mapping overview

There are two different styles of C++ mapping for attributes, depending on the `use` value in the attribute declaration:

- *Optional attributes*—if an attribute is declared with `use="optional"` (or if the `use` setting is omitted altogether), the generated `get`*`Attribute`*`()` function returns a pointer, instead of a reference, to the attribute value. This enables you to test whether the attribute is set or not by testing the pointer for nilness (whether it equals 0).
- *Required attributes*—if an attribute is declared with `use="required"`, the generated `get`*`Attribute`*`()` function returns a reference to the attribute value.

Optional attribute example

Example 158 shows how to define a sequence type with a single optional attribute, `prop`, of `xsd:string` type (attributes are optional by default).

Example 158: *Definition of a Sequence Type with an Optional Attribute*

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

C++ mapping for an optional attribute

Example 159 shows an outline of the C++ `SequenceType` class generated from Example 158, which defines accessor and modifier functions for the optional `prop` attribute.

Example 159: *Mapping an Optional Attribute to C++*

```
// C++
class SequenceType : public IT_Bus::SequenceComplexType
{
  public:
    SequenceType();
    ...
```

Example 159: *Mapping an Optional Attribute to C++*

```
1       const IT_Bus::String * getprop() const;
        IT_Bus::String * getprop();

2       void setprop(const IT_Bus::String * val);
3       void setprop(const IT_Bus::String & val);
    };
```

The preceding C++ mapping can be explained as follows:

1. If the attribute is set, returns a pointer to its value; if not, returns `0`.
2. If `val != 0`, sets the attribute to `*val` (makes a copy); if `val == 0`, unsets the attribute.
3. Sets the attribute to `val` (makes a copy). This is a convenience function that enables you to set the attribute without using a pointer.

Required attribute example

Example 160 shows how to define a sequence type with a single required attribute, `prop`, of `xsd:string` type.

Example 160: *Definition of a Sequence Type with a Required Attribute*

```
<complexType name="SequenceType">
    <sequence>
        <element name="varFloat" type="xsd:float"/>
        <element name="varInt" type="xsd:int"/>
        <element name="varString" type="xsd:string"/>
    </sequence>
    <attribute name="prop" type="xsd:string" use="required"/>
</complexType>
```

C++ mapping for a required attribute

Example 161 shows an outline of the C++ `SequenceType` class generated from Example 160 on page 318, which defines accessor and modifier functions for the required `prop` attribute.

Example 161: *Mapping a Required Attribute to C++*

```
// C++
class SequenceType : public IT_Bus::SequenceComplexType
{
  public:
    SequenceType();
    ...
    const IT_Bus::String & getprop() const;
    IT_Bus::String & getprop();

    void setprop(const IT_Bus::String & val);
};
```

In this case, the `getprop()` accessor function returns a *reference* to a string (that is, `IT_Bus::String&`), rather than a pointer to a string.

Limitations

The following attribute types are *not* supported:

- `xsd:IDREFS`
- `xsd:ENTITY`
- `xsd:ENTITIES`
- `xsd:NOTATION`
- `xsd:NMTOKEN`
- `xsd:NMTOKENS`

Attribute Groups

An attribute group, which is defined using the `attributeGroup` element, is a convenient shortcut that enables you to reference a group of attributes in user-defined complex types. The `attributeGroup` element is used in two distinct ways: for defining an attribute group and for referencing an existing attribute group.

To define a new attribute group (which should be done within the scope of a `schema` element), use the following syntax:

```
<attributeGroup
  name="AttrGroup_NCName">
    <attribute ... > ... </attribute>
    ...
    <attributeGroup ref="..." ... > ... </attributeGroup>
    ...
</attributeGroup>
```

To reference an existing attribute from within a complex type definition, use the following syntax:

```
<attributeGroup ref="AttrGroup_QName" />
```

Note: Attribute groups are currently supported only by the SOAP binding.

Simple attribute groups

Example 162 shows how to define an attribute group, DimAttrGroup, which contains three attributes, length, breadth, and height, and is referenced by the complex type, Package.

Example 162:*Example of Defining a Simple Attribute Group*

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:tns="http://schemas.iona.com/attr_example"
targetNamespace="http://schemas.iona.com/attr_example">

    <attributeGroup name="DimAttrGroup">
        <attribute name="length" type="xsd:int"/>
        <attribute name="breadth" type="xsd:int"/>
        <attribute name="height" type="xsd:int"/>
    </attributeGroup

    <complexType name="Package">
        <sequence> ... </sequence>
        <attributeGroup ref="tns:DimAttrGroup" />
    </complexType>

</schema>
```

The preceding Package type defined in Example 162 on page 320 is exactly equivalent to the Package type defined in Example 163. In other words, referencing an attribute group has essentially the same effect as defining the attributes directly within the type.

Example 163:*Equivalent Type Using Attributes instead of Attribute Group*

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:tns="http://schemas.iona.com/attr_example"
 targetNamespace="http://schemas.iona.com/attr_example">

    <complexType name="Package">
        <sequence> ... </sequence>
        <attribute name="length" type="xsd:int"/>
        <attribute name="breadth" type="xsd:int"/>
        <attribute name="height" type="xsd:int"/>
    </complexType>

</schema>
```

Nested attribute groups

It is also possible to nest attribute groups by referencing an attribute group within another attribute group definition. Example 164 shows how to define an attribute group, `DimAndColor`, which recursively references another attribute group, `DimAttrGroup`.

Example 164: *Example of Defining a Nested Attribute Group*

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://schemas.iona.com/attr_example"
  targetNamespace="http://schemas.iona.com/attr_example>

    <attributeGroup name="DimAttrGroup">
        <attribute name="length" type="xsd:int"/>
        <attribute name="breadth" type="xsd:int"/>
        <attribute name="height" type="xsd:int"/>
    </attributeGroup

    <attributeGroup name="DimAndColor">
        <attributeGroup ref="tns:DimAttrGroup"/>
        <attribute name="Color" type="xsd:string"/>
    </attributeGroup>

</schema>
```

C++ mapping

The C++ mapping for a type that references an attribute group is precisely the same as if the attributes were defined directly within the type. In other words, all of the attribute groups are recursively unwrapped and the attributes are inserted directly into the type definition. The type is then mapped to C++ according to the usual mapping rules.

For details of the C++ mapping of attributes, see "Attributes" on page 316.

Nesting Complex Types

It is possible to nest complex types within each other. When mapped to C++, the nested complex types map to a nested hierarchy of classes, where each instance of a nested type is stored in a member variable of its containing class.

Avoiding anonymous types

In general, it is a good idea to name types that are nested inside other types, instead of using anonymous types. This results in simpler code when the types are mapped to C++.

For an example of the recommended style of declaration, with a named nested type, see Example 165.

WSDL example

Example 165 shows an example of a nested complex type, which features a choice complex type, `NestedChoiceType`, nested inside a sequence complex type, `SeqOfChoiceType`.

Example 165:*Definition of Nested Complex Type*

```
<schema targetNamespace="http://soapinterop.org/xsd"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <complexType name="NestedChoiceType">
        <choice>
            <element name="varFloat" type="xsd:float"/>
            <element name="varInt" type="xsd:int"/>
        </choice>
    </complexType>
    <complexType name="SeqOfChoiceType">
        <sequence>
         <element name="varString" type="xsd:string"/>
         <element name="varChoice" type="xsd1:NestedChoiceType"/>
        </sequence>
    </complexType>
    ...
</schema>
```

C++ mapping of NestedChoiceType

The XML schema choice complex type, `NestedChoiceType`, is a simple choice complex type, which is mapped to C++ in the standard way. Example 166 shows an outline of the generated C++ `NestedChoiceType` class.

Example 166:*Mapping of NestedChoiceType to C++*

```
// C++
class NestedChoiceType : public IT_Bus::ChoiceComplexType
{
    ...
  public:
    NestedChoiceType();
    NestedChoiceType(const NestedChoiceType& copy);
    virtual ~NestedChoiceType();

    virtual const IT_Bus::QName &     get_type() const ;

    NestedChoiceType& operator= (const NestedChoiceType&
   assign);

    const IT_Bus::Float getvarFloat() const;
    void setvarFloat(const IT_Bus::Float& val);

    const IT_Bus::Int getvarInt() const;
    void setvarInt(const IT_Bus::Int& val);

    IT_Bus::UInt get_discriminator() const;

  private:
```

Example 166:*Mapping of NestedChoiceType to C++*

```
    ...
};
```

C++ mapping of SeqOfChoiceType

The XML schema sequence complex type, `SeqOfChoiceType`, has the `NestedChoiceType` nested inside it. Example 167 shows an outline of the generated C++ `SeqOfChoiceType` class, which shows how the nested complex type is mapped within a sequence complex type.

Example 167:*Mapping of SeqOfChoiceType to C++*

```
// C++
class SeqOfChoiceType : public IT_Bus::SequenceComplexType
{
    ...
  public:
    SeqOfChoiceType();
    SeqOfChoiceType(const SeqOfChoiceType& copy);
    virtual ~SeqOfChoiceType();
    ...
    virtual const IT_Bus::QName & get_type() const;

    SeqOfChoiceType& operator= (const SeqOfChoiceType& assign);

    const IT_Bus::String & getvarString() const;
    IT_Bus::String & getvarString();
    void setvarString(const IT_Bus::String & val);

    const NestedChoiceType & getvarChoice() const;
    NestedChoiceType & getvarChoice();
    void setvarChoice(const NestedChoiceType & val);

  private:
    ...
};
```

The nested type, `NestedChoiceType`, can be accessed and modified using the `getvarChoice()` and `setvarChoice()` functions respectively.

Example

Consider a port type that defines an `echoSeqOfChoice` operation. The `echoSeqOfChoice` operation takes a `SeqOfChoiceType` type as an in parameter and then echoes this value in the response. Example 157 shows how a client could use a proxy instance, `bc`, to invoke the `echoSeqOfChoice` operation.

Example 168:*Client Invoking an echoSeqOfChoice Operation*

```
// C++
NestedChoiceType nested;
nested.setvarFloat(3.14159);

SeqOfChoiceType seqIn, seqResult;
seqIn.setvarChoice(nested);
seqIn.setvarString("You can use a string constant here.");
try {
    bc.echoSeqOfChoice(seqIn, seqResult);

    if(
   (seqResult.getvarString().compare(seqIn.getvarString()) != 0) ||
   (seqResult.getvarChoice().get_discriminator()
        !=seqIn.getvarChoice().get_discriminator()))
    {
        cout << endl << "echoSeqOfChoice FAILED" << endl;
        return;
    }
} catch (IT_Bus::FaultException &ex)
{
    cout << "Caught Unexpected FaultException" << endl;
    cout << ex.get_description().c_str() << endl;
}
```

Deriving a Complex Type from a Simple Type

Artix supports derivation of a complex type from a simple type, for which the following kinds of derivation are supported:

- Derivation by restriction.
- Derivation by extension.

A simple type has, by definition, neither sub-elements nor attributes. Hence, one of the main reasons for deriving a complex type from a simple type is to add attributes to the simple type (derivation by extension).

Derivation by restriction

Example 169 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 169:*Deriving a Complex Type from a Simple Type by Restriction*

```
<xsd:complexType name="orderNumber">
    <xsd:simpleContent>
        <xsd:restriction base="xsd:decimal">
            <xsd:maxExclusive value="1000000"/>
        </xsd:restriction>
    </xsd:simpleContent>
</xsd:complexType>
```

The `<simpleContent>` tag indicates that the new type does not contain any sub-elements and the `<restriction>` tag defines the derivation by restriction from `xsd:decimal`.

Derivation by extension

Example 170 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 170:*Deriving a Complex Type from a Simple Type by Extension*

```
<xsd:complexType name="internationalPrice">
    <xsd:simpleContent>
        <xsd:extension base="xsd:decimal">
            <xsd:attribute name="currency" type="xsd:string"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
```

The `<simpleContent>` tag indicates that the new type does not contain any sub-elements and the `<extension>` tag defines the derivation by extension from `xsd:decimal`.

C++ mapping

Example 171 shows an outline of the C++ `internationalPrice` class generated from Example 170 on page 325.

Example 171: *Mapping the internationalPrice Type to C++*

```
// C++
class internationalPrice : public IT_Bus::SimpleContentComplexType
{
    ...
  public:
    internationalPrice();
    internationalPrice(const internationalPrice& copy);
    virtual ~internationalPrice();

    ...
    virtual const IT_Bus::QName & get_type() const;

    internationalPrice& operator= (const internationalPrice& assign);

    const IT_Bus::String & getcurrency() const;
    IT_Bus::String & getcurrency();
    void setcurrency(const IT_Bus::String & val);

    const IT_Bus::Decimal & get_simpleTypeValue() const;
    IT_Bus::Decimal & get_simpleTypeValue();
    void set_simpleTypeValue(const IT_Bus::Decimal & val);
    ...
};
```

The value of the currency attribute, which is added by extension, can be accessed and modified using the `getcurrency()` and `setcurrency()` member functions. The simple type value (that is, the value enclosed between the `<internationalPrice>` and `</internationalPrice>` tags) can be accessed and modified by the `get_simpleTypeValue()` and `set_simpleTypeValue()` member functions.

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.

This subsection describes the C++ mapping for complex types derived from complex types and, in particular, describes the coding pattern for calling a function either with base type arguments or with derived type arguments.

Allowed inheritance relationships

Figure 29 shows the inheritance relationships allowed between complex types. As well as inheriting between the same kind of complex type (sequence from sequence, choice from choice, and all from all), derivation by extension also supports cross-inheritance. For example, a sequence can derive from a choice, a choice from an all, an all from a choice, and so on.

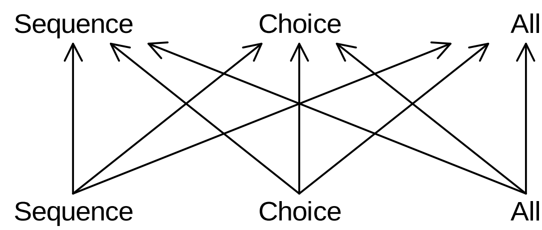

Figure 29: *Allowed Inheritance Relationships for Complex Types*

Derivation by restriction

Example 172 shows an example of deriving a sequence from a sequence by restriction. In this example, `RestrictedStruct` is derived from `SimpleStruct` by restriction. The standard tag used to declare inheritance by restriction is <restriction base="*BaseComplexType*"/>.

Example 172:*Example of Deriving a Sequence by Restriction*

```
// C++
<complexType name="SimpleStruct">
    <sequence>
        <element name="varFloat" type="float"/>
        <element name="varInt" type="int"/>
        <element name="varString" type="string"/>
    </sequence>
    <attribute name="varAttrString" type="string"/>
</complexType>
...
1 <complexType name="RestrictedStruct">
    <complexContent>
2        <restriction base="tns:SimpleStruct">
3            <sequence>
                <element name="varFloat" type="float"/>
                <element name="varInt" type="int"/>
4              <element name="varString" type="string"
                        fixed="Restricted"/>
            </sequence>
        </restriction>
    </complexContent>
</complexType>
```

The preceding type definition can be explained as follows:

1. This <complexType> tag introduces the definition of the derived sequence type, RestrictedStruct.
2. The <restriction> tag indicates that this type derives by restriction from the SimpleStruct type.
3. Elements that appear in the SimpleStruct base type must be duplicated here, if they are to be included in the derived type, but they can also have extra restrictions imposed on them.
4. The varString element is restricted here to have the fixed value, Restricted.

Derivation by extension

Example 173 shows an example of deriving a sequence from a sequence by extension. In this example, DerivedStruct_BaseStruct is derived from SimpleStruct by extension. The standard tag used to declare inheritance by extension is <extension base="*BaseComplexType*"/>.

Example 173: *Example of Deriving a Sequence by Extension*

```
  <complexType name="SimpleStruct">
      <sequence>
          <element name="varFloat" type="float"/>
          <element name="varInt" type="int"/>
          <element name="varString" type="string"/>
      </sequence>
      <attribute name="varAttrString" type="string"/>
  </complexType>
  ...
1 <complexType name="DerivedStruct_BaseStruct">
2     <complexContent mixed="false">
3         <extension base="tns:SimpleStruct">
4             <sequence>
                  <element name="varStringExt" type="string"/>
                  <element name="varFloatExt" type="float"/>
              </sequence>
5             <attribute name="attrString1" type="string"/>
          </extension>
      </complexContent>
  </complexType>
```

The preceding type definition can be explained as follows:

1. This <complexType> tag introduces the definition of the derived sequence type, DerivedStruct_BaseStruct.
2. The <complexContent> tag indicates that what follows is a declaration of contained tags. The mixed="false" setting indicates that the type can contain only tags, not text.
3. The <extension> tag indicates that this type derives by extension from the SimpleStruct type.
4. The <sequence> tag defines extra type members that are specific to the derived type, DerivedStruct_BaseStruct.
5. You can also declare attributes specific to the derived type.

C++ mapping for derivation by restriction

The C++ mapping for derivation by restriction is essentially the same as the C++ mapping for derivation by extension.

In the case of derivation by restriction, however, Artix does not enforce all of the restrictions at runtime. To ensure interoperability, therefore, your service should enforce the restrictions declared in the WSDL contract.

C++ mapping for derivation by extension

The sequence types defined in Example 173 on page 328, `SimpleStruct` and `DerivedStruct_BaseStruct`, map to C++ as shown in Example 174.

Example 174: *C++ Mapping of a Derived Sequence Type*

```
// C++
class SimpleStruct : public IT_Bus::SequenceComplexType
{
  public:
    static const IT_Bus::QName type_name;

    SimpleStruct();
    ...
    IT_Bus::AnyType &
    operator=(const IT_Bus::AnyType & rhs);

    SimpleStruct &
    operator=(const SimpleStruct & rhs);

    const SimpleStruct * get_derived() const;
    virtual IT_Bus::AnyType::Kind get_kind() const;
    virtual const IT_Bus::QName & get_type() const;
    ...
    IT_Bus::Float       getvarFloat();
    const IT_Bus::Float getvarFloat() const;
    void setvarFloat(const IT_Bus::Float val);

    IT_Bus::Int       getvarInt();
    const IT_Bus::Int getvarInt() const;
    void setvarInt(const IT_Bus::Int val);

    IT_Bus::String &       getvarString();
    const IT_Bus::String & getvarString() const;
    void setvarString(const IT_Bus::String & val);

    IT_Bus::String &       getvarAttrString();
    const IT_Bus::String & getvarAttrString() const;
    void setvarAttrString(const IT_Bus::String & val);

  private:
    ...
};

typedef IT_AutoPtr<SimpleStruct> SimpleStructPtr;
```

Example 174: *C++ Mapping of a Derived Sequence Type*

```
...
class IT_TEST_WSDL_API DerivedStruct_BaseStruct : public SimpleStruct,
   public virtual IT_Bus::ComplexContentComplexType
{
  public:
    static const IT_Bus::QName type_name;

    DerivedStruct_BaseStruct();
    DerivedStruct_BaseStruct(const DerivedStruct_BaseStruct & copy);
    virtual ~DerivedStruct_BaseStruct();
    ...
    IT_Bus::String &        getvarStringExt();
    const IT_Bus::String & getvarStringExt() const;
    void setvarStringExt(const IT_Bus::String & val);

    IT_Bus::Float           getvarFloatExt();
    const IT_Bus::Float     getvarFloatExt() const;
    void setvarFloatExt(const IT_Bus::Float val);

    IT_Bus::String &        getattrString1();
    const IT_Bus::String & getattrString1() const;
    void setattrString1(const IT_Bus::String & val);

  private:
    ...
};
```

The C++ `DerivedStruct_BaseStruct` class derives directly from the C++ `SimpleStruct` class. Hence, all of the accessors and modifiers declared in the base class, `SimpleStruct`, are also available to the derived class, `DerivedStruct_BaseStruct`.

Using a base type as a holder

The `SimpleStruct` type declared in Example 174 on page 329 is really a dual-purpose type. That is, a `SimpleStruct` instance can be used in one of the following different ways:

- As a `SimpleStruct` data type (base type)—member data is accessed by invoking `get`*ElementName*`()` and `set`*ElementName*`()` functions directly on the `SimpleStruct` instance.
- As a holder type (derived type holder)—in this usage pattern, the `SimpleStruct` instance is used to hold a reference to a more derived type (for example, `DerivedStruct_BaseStruct`).

Holder type functions

If you are using `SimpleStruct` as a holder type, the following member functions are relevant:

- `SimpleStruct(const SimpleStruct & copy)`—the `SimpleStruct` copy constructor is used to initialize the reference held by the `SimpleStruct` holder object. The type passed to the copy constructor can be any type derived from `SimpleStruct`.

- `SimpleStruct & operator=(const SimpleStruct & rhs)`—alternatively, if you already have a `SimpleStruct` object, you can change the reference held by making an assignment to the `SimpleStruct` holder.
- `const SimpleStruct * get_derived() const`—if you want to access the derived type held by a `SimpleStruct` holder object, call the `get_derived()` member function and then dynamically cast the return value to the appropriate type.
- `const IT_Bus::QName & get_type() const`—call `get_type()` to get the QName of the derived type held by a `SimpleStruct` holder object.

Polymorphism

When a WSDL operation is defined to take arguments of a base class type (for example, `SimpleStruct`), it is also possible to send and receive arguments of a type derived from that base class (for example, `DerivedStruct_BaseStruct`).

For reasons of backward compatibility, however, the C++ code required for calling an operation with derived type arguments is different from the C++ code required for calling an operation with base type arguments.

Sample WSDL operation

For example, consider the definition of the following WSDL operation, `test_SimpleStruct`, that takes an *in* argument of `SimpleStruct` type and returns an *out* argument of `SimpleStruct` type.

Example 175:*The test_SimpleStruct Operation with Base Type Arguments*

```
...
<message name="test_SimpleStruct">
    <part name="x" element="tns:SimpleStruct_x"/>
</message>
<message name="test_SimpleStruct_response">
    <part name="return" element="tns:SimpleStruct_return"/>
</message>
...
<operation name="test_SimpleStruct">
  <input name="test_SimpleStruct"
         message="tns:test_SimpleStruct"/>
  <output name="test_SimpleStruct_response"
          message="tns:test_SimpleStruct_response"/>
</operation>
```

The preceding `test_SimpleStruct` WSDL operation maps to the following C++ function (in the `TypeTestClient` client proxy class).

```
// C++
virtual void
test_SimpleStruct(
    const SimpleStruct &x,
    SimpleStruct &_return,
) IT_THROW_DECL((IT_Bus::Exception));
```

To call the preceding `test_SimpleStruct()` function in C++, use one of the following programming patterns, depending on the type of arguments passed:

- Base or derived type arguments.
- Base type arguments only (for legacy code).

Base or derived type arguments

Example 176 shows you how to call the `test_SimpleStruct()` function with derived type arguments (of `DerivedStruct_BaseStruct` type). Generally, this coding pattern can be used to pass either base type or derived type arguments.

Example 176:*Calling test_SimpleStruct() with Derived Type Arguments*

```
  // C++
1 DerivedStruct_BaseStruct x;

  // Base members
2 x.setvarFloat((IT_Bus::Float) 3.14);
  x.setvarInt((IT_Bus::Int) 42);
  x.setvarString((IT_Bus::String) "BaseStruct-x");
  x.setvarAttrString((IT_Bus::String) "BaseStructAttr-x");
  // Derived members
  x.setvarFloatExt((IT_Bus::Float) -3.14f);
  x.setvarStringExt((IT_Bus::String) "DerivedStruct-x");
  x.setattrString1((IT_Bus::String) "DerivedAttr-x");

3 SimpleStruct x_holder(x);
4 SimpleStruct ret_holder;

5 proxy->test_SimpleStruct(x_holder, ret_holder);

6 const DerivedStruct_BaseStruct* ret_derived
      = dynamic_cast<const DerivedStruct_BaseStruct*>(
            ret_holder.get_derived()
        );

  // Use ret_derived type value...
  ...
```

The preceding C++ code can be explained as follows:

1. The in parameter, `x`, of the `test_SimpleStruct()` function is declared to be of derived type, `DerivedStruct_BaseStruct`.
2. Both the base members and the derived members of the *in* parameter, `x`, are initialized here.

3. The derived type, x, is wrapped by a base type instance, x_holder. In this case, the SimpleStruct object, x_holder, is used purely as a holder type; x_holder does *not* directly represent a SimpleStruct type argument.
4. The return type, ret_holder, is declared to be of SimpleStruct type. Here also, ret_holder is treated as a holder type.
5. Call the remote test_SimpleStruct() function, passing in the two holder instances, x_holder and ret_holder.
6. To obtain a pointer to the derived type return value, call SimpleStruct::get_derived(). This function returns a pointer to the derived type contained in the ret_holder object. You can then cast the returned pointer to the appropriate type using the dynamic_cast<> operator.

   If necessary, you can call the SimpleStruct::get_type() function to discover the QName of the returned type before attempting to cast the return value.

Base type arguments only (for legacy code)

Example 177 shows you how to call the test_SimpleStruct() function with base type arguments (of SimpleStruct type). This coding pattern is supported for reasons of backward compatibility.

Example 177: *Calling test_SimpleStruct() with Base Type Arguments*

```
  // C++
1 SimpleStruct x;

  // Base members
2 x.setvarFloat((IT_Bus::Float) 3.14);
  x.setvarInt((IT_Bus::Int) 42);
  x.setvarString((IT_Bus::String) "BaseStruct-x");
  x.setvarAttrString((IT_Bus::String) "BaseStructAttr-x");

3 SimpleStruct ret;

4 proxy->test_SimpleStruct(x, ret);

  // Use ret value...
  cout << ret.getvarFloat();
  ...
```

The preceding C++ code can be explained as follows:

1. The in parameter, x, of the test_SimpleStruct() function is declared to be of base type, SimpleStruct.
2. The members of the SimpleStruct *in* parameter, x, are initialized.
3. The return value, ret, of the test_SimpleStruct() function is declared to be of base type, SimpleStruct.

> **Note:** The return value must be allocated *before* calling the test_SimpleStruct() function.

4. This line calls the remote `test_SimpleStruct()` function with in parameter, `x`, and return parameter, `ret`.

   Note: In this example, it is assumed that the return value is of base type, `SimpleStruct`. In general, however, the return type might be of derived type (see “Base or derived type arguments” on page 332).

Arrays

This subsection describes how to define and use basic Artix array types. In addition to these basic array types, Artix also supports SOAP arrays, which are discussed in “SOAP Arrays” on page 400.

Array definition syntax

An 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 to IT_Bus::ArrayT

When a sequence complex type declaration satisfies the special conditions to be an array, it is mapped to C++ differently from a regular sequence complex type. Instead of mapping to `IT_Bus::SequenceComplexType`, the array maps to the `IT_Bus::ArrayT<ElementType>` template type. Effectively, the C++ array template class can be treated like a vector.

For example, the mapped C++ array class supports the `size()` member function and individual elements can be accessed using the `[]` operator.

WSDL array example

Example 178 shows how to define a one-dimensional string array, `ArrayOfString`, whose size can lie anywhere in the range `0` to unbounded.

Example 178:*Definition of an Array of Strings*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >
    <types>
        <schema ... >
           <complexType name="ArrayOfString">
               <sequence>
                   <element name="varString" type="xsd:string"
                            minOccurs="0" maxOccurs="unbounded"/>
               </sequence>
           </complexType>
             ...
    ...
</definitions>
```

C++ mapping

Example 179 shows how the `ArrayOfString` string array (from Example 178 on page 335) maps to C++.

Example 179:*Mapping of ArrayOfString to C++*

```
// C++
class ArrayOfString : public IT_Bus::ArrayT<IT_Bus::String>
{
  public:
    ArrayOfString();
    ArrayOfString(size_t dimension0);
    ArrayOfString(const ArrayOfString& copy);
    virtual ~ArrayOfString();

    virtual const IT_Bus::QName & get_type() const;

    ArrayOfString& operator= (const IT_Vector<IT_Bus::String>& assign);

    const IT_Bus::ElementListT<IT_Bus::String> & getvarString() const;

    IT_Bus::ElementListT<IT_Bus::String> & getvarString();

    void setvarString(const IT_Bus::ElementListT<IT_Bus::String> & val);

};

typedef IT_AutoPtr<ArrayOfString> ArrayOfStringPtr;
```

Notice that the C++ array class provides accessor functions, `getvarString()` and `setvarString()`, just like any other sequence complex type with occurrence constraints (see “Sequence Occurrence Constraints” on page 369). The accessor functions are superfluous, however, because the array's elements are more easily accessed by invoking vector operations directly on the `ArrayOfString` class.

Example

Example 180 shows an example of how to allocate and initialize an `ArrayOfString` instance, by treating it like a vector (for a complete list of vector operations, see "Summary of IT_Vector Operations" on page 410).

Example 180:*Example for a One-Dimensional Array*

```
// C++
// Array of String
ArrayOfString a(4);

a[0] = "One";
a[1] = "Two";
a[2] = "Three";
a[3] = "Four";
```

Multi-dimensional arrays

You can define multi-dimensional arrays by nesting array definitions (see "Nesting Complex Types" on page 321 for a discussion of nested types). Example 181 shows an example of how to define a two-dimensional string array, `ArrayOfArrayOfString`.

Example 181:*Definition of a Multi-Dimensional String Array*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >
    <types>
        <schema ... >
           <complexType name="ArrayOfString">
               <sequence>
                   <element name="varString" type="xsd:string"
                           minOccurs="0" maxOccurs="unbounded"/>
               </sequence>
           </complexType>
           <complexType name="ArrayOfArrayOfString">
               <sequence>
                   <element name="nestArray"
                           type="xsd1:ArrayOfString"
                           minOccurs="0" maxOccurs="unbounded"/>
               </sequence>
           </complexType>
             ...
    ...
</definitions>
```

Both the nested array type, `ArrayOfArrayOfString`, and the sub-array type, `ArrayOfString`, must conform to the standard array definition syntax. Multi-dimensional arrays can be nested to an arbitrary degree, but each sub-array must be a named type (that is, anonymous nested array types are not supported).

Example for multidimensional array

Example 182 shows an example of how to allocate and initialize a multi-dimensional array, of `ArrayOfArrayOfString` type.

Example 182:*Example for a Multi-Dimensional Array*

```
// C++
// Array of array of String
ArrayOfArrayOfString a2(2);

for (int i = 0 ; i < a2.size(); i++) {
    a2[i].set_size(2);
}

a2[0][0] = "ZeroZero";
a2[0][1] = "ZeroOne";
a2[1][0] = "OneZero";
a2[1][1] = "OneOne";
```

The `set_size()` function enables you to set the dimension of each sub-array individually. If you choose different sizes for the sub-arrays, you can create `a2` as a ragged two-dimensional array.

Automatic conversion to IT_Vector

In general, a multi-dimensional array can automatically convert to a vector of `IT_Vector`<*SubArray*> type, where *SubArray* is the array element type.

Example 183 shows how an instance, `a2`, of `ArrayOfArrayOfString` type converts to an instance of `IT_Vector<ArrayOfString>` type by assignment.

Example 183:*Converting a Multi-Dimensional Array to IT_Vector Type*

```
// Array of array of String
ArrayOfArrayOfString a2(2);

for (int i = 0 ; i < a2.size(); i++) {
    a2[i].set_size(2);
}
...
// Obtain reference to the underlying IT_Vector type
IT_Vector<ArrayOfString>& v_a2 = a2;

cout << v_a2[0][0] << " " << v_a2[0][1] << " "
     << v_a2[1][0] << " " << v_a2[1][1] << endl;
cout << "v_a2.size() = " << v_a2.size() << endl;
```

References

For more details about vector types see:

- The "IT_Vector Template Class" on page 408.
- The section on C++ ANSI vectors in *The C++ Programming Language*, third edition, by Bjarne Stroustrup.

Model Group Definitions

A model group definition is a convenient shortcut that enables you to reference a group of elements from a user-defined complex type.

- To define a new model group (which should be done within the scope of a `schema` element), use the following syntax:

```
<group
  name="Group_NCName">
    [<sequence> | <choice> ]
        ...
    [</sequence> | </choice> ]
</group>
```

- To reference an existing model group from within a complex type definition or from within another model group definition, use the following syntax:

```
<group ref="Group_QName"/>
```

Note: Model groups are currently supported only by the SOAP binding.

Group of sequence

Example 184 shows how to define a model group, `PassengerName`, which contains a sequence of elements.

Example 184: *Model Group Definition Containing a Sequence*

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://schemas.iona.com/group"
        targetNamespace="http://schemas.iona.com/group">

    <group name="PassengerName">
        <sequence>
            <element name="FirstName" type="xsd:string"/>
            <element name="SecondName" type="xsd:string"/>
        </sequence>
    </group>

</schema>
```

When the preceding XSD schema is mapped to C++, the PassengerName model group is mapped to its own C++ class, PassengerName, as shown in Example 185.

Example 185:*PassengerName Model Group Mapping to C++*

```
// C++
class PassengerName : public IT_Bus::SequenceComplexType
{
  public:
    ...
    PassengerName();
    PassengerName(const PassengerName & copy);
    virtual ~PassengerName();
    ...
    IT_Bus::String &       getFirstName();
    const IT_Bus::String & getFirstName() const;
    void setFirstName(const IT_Bus::String & val);

    IT_Bus::String &       getSecondName();
    const IT_Bus::String & getSecondName() const;
    void setSecondName(const IT_Bus::String & val);

  private:
    ...
};
```

Group of choice

Example 186 shows how to define a model group, PassengerID, which contains a choice of elements.

Example 186:*Model Group Definition Containing a Choice*

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://schemas.iona.com/group"
        targetNamespace="http://schemas.iona.com/group">

    <group name="PassengerID">
        <choice>
            <element name="PassportNo" type="xsd:integer"/>
            <element name="IDCardNo" type="xsd:integer"/>
        </choice>
    </group>

</schema>
```

When the preceding XSD schema is mapped to C++, the PassengerID model group is mapped to a C++ class, PassengerID, in just the same way as a regular choice complex type (see, for example, “Choice Complex Types” on page 311).

Recursive group references

Example 187 shows how to define a model group, `Hop`, which recursively references another model group definition, `PassengerName`.

Example 187: *Model Group Definition with Recursive Reference*

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://schemas.iona.com/group"
        targetNamespace="http://schemas.iona.com/group">

    <group name="PassengerName">
        <sequence>
            <element name="FirstName" type="xsd:string"/>
            <element name="SecondName" type="xsd:string"/>
        </sequence>
    </group>

    <group name="Hop">
        <sequence>
            <group ref="tns:PassengerName"/>
            <element name="origin" type="xsd:string"/>
            <element name="destination" type="xsd:string"/>
        </sequence>
    </group>

</schema>
```

When the preceding XSD schema is mapped to C++, the `Hop` model group maps to a C++ class, `Hop`, like a regular sequence complex type. In particular, the recursive reference to another model group, `tns:PassengerName`, is mapped to a pair of accessor and modifier functions, `getPassengerName()` and `setPassengerName()`, as shown in Example 188.

Example 188: *Hop Model Group Mapping to C++*

```
// C++
class Hop : public IT_Bus::SequenceComplexType
{
  public:
    ...
    Hop();
    Hop(const Hop & copy);
    virtual ~Hop();
    ...
    PassengerName &       getPassengerName();
    const PassengerName & getPassengerName() const;
    void setPassengerName(const PassengerName & val);

    IT_Bus::String &       getorigin();
    const IT_Bus::String & getorigin() const;
    void  setorigin(const IT_Bus::String & val);

    IT_Bus::String &       getdestination();
    const IT_Bus::String & getdestination() const;
    void setdestination(const IT_Bus::String & val);
```

Example 188: *Hop Model Group Mapping to C++*

```
  private:
    ...
};
```

Repeated group references

Example 189 shows how to define a model group, `TwoHops`, which references the `Hop` model group twice.

Example 189: *Model Group Definition with Repeated References*

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://schemas.iona.com/group"
        targetNamespace="http://schemas.iona.com/group">

    <group name="TwoHops">
        <sequence>
            <group ref="tns:Hop"/>
            <group ref="tns:Hop"/>
        </sequence>
    </group>

</schema>
```

When the preceding XSD schema is mapped to C++, the `TwoHops` model group maps to a C++ class, `TwoHops`, as shown in Example 190.

Example 190: *TwoHops Model Group Mapping to C++*

```
// C++
class TwoHops : public IT_Bus::SequenceComplexType
{
  public:
    ...
    TwoHops();
    TwoHops(const TwoHops & copy);
    virtual ~TwoHops();
    ...
    Hop &        getHop();
    const Hop & getHop() const;
    void setHop(const Hop & val);

    Hop &        getHop_1();
    const Hop & getHop_1() const;
    void setHop_1(const Hop & val);

  private:
    ...
};
```

Two sets of accessors and modifiers are generated: the first model group reference maps to the functions, `getHop()` and `setHop()`; the second model group reference maps to the functions, `getHop_1()` and `setHop_1()`.

In general, an `N+1`th repetition of a model group reference would generate a pair of functions, `getHop_N()` and `setHop_N()`.

Binary Types and MTOM

This section describes how to use the schema binary types, `xs:base64Binary` and `xs:hexBinary`, in the context of the MTOM protocol.

Introduction to MTOM

The Message Transmission Optimization Mechanism (MTOM) is a protocol designed to optimize the transmission of binary data within SOAP 1.2 messages. When MTOM is enabled, it converts SOAP messages into MIME multipart/related messages, where the binary data from the SOAP message is transmitted as a series of MIME attachments.

MTOM is incompatible with certain WS-Security features, so it is recommended that you disable MTOM when security is enabled.

Advantages of MTOM

MTOM offers the following advantages:

- *Optimization*—raw binary data can be written *directly* into the MIME multipart/related message, skipping the conversion of raw binary to base-64 encoding (or raw binary to hexadecimal encoding). This leads to faster marshalling and smaller message sizes.
- *MIME content type*—a MIME content type can be associated with the binary data and the content type can be accessed from the application code.
- *Ease-of-use*—MTOM is easy to enable (particularly in comparison to the SOAP-with-Attachments standard).

Specifications

The following W3 specifications are relevant to MTOM:

- SOAP Message Transmission Optimization Mechanism (MTOM).
- XML-binary Optimized Packaging (XOP).
- Describing Media Content of Binary Data in XML (XMIME).

Enabling MTOM

The MTOM optimization is supported only by SOAP 1.2. You can enable the optimization as follows:

- *Client side*—to enable MTOM on the client side, add the following setting to your application's scope in the Artix configuration file:

```
plugins:soap12:enable_mtom_serialization = "true";
```

- *Server side*—MTOM is *always* enabled on the server side. If a server detects that an incoming SOAP message conforms to MTOM, it will automatically apply MTOM to decode the message.

Default XOP Encoding

The simplest approach to using MTOM is where you enable MTOM on the client side and leave the WSDL contract unchanged. In this case, MTOM automatically chooses the default XOP encoding for any binary types that it encounters in the WSDL (that is, `xs:base64Binary`, `xs:hexBinary`, and any types derived from them).

WSDL example

Example 191 shows the definition of a `data` schema element that contains two elements, `photo` and `sig`, of `xs:base64Binary` type. This is a standard schema example—there are no MTOM-specific details in it.

Example 191:*WSDL Example for Default XOP Encoding*

```
<wsdl:types>
  <xs:schema targetNamespace="http://example.og/stuff"
          xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="data">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="photo" type="xs:base64Binary"/>
          <xs:element name="sig" type="xs:base64Binary"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    ...
  </xs:schema>
</wsdl:types>
```

Plain SOAP message

Example 192 shows an example of a plain, *non-MTOM* SOAP message containing binary data (using the `data` element defined in Example 191). Both the `m:photo` element and the `m:sig` element contain binary data encoded using base-64 encoding and embedded directly in the elements themselves.

Example 192:*Plain SOAP Message Containing Binary Data*

```
<soap:Envelope
    xmlns:soap='http://www.w3.org/2003/05/soap-envelope'
    xmlns:xmlmime='http://www.w3.org/2004/11/xmlmime'>
  <soap:Body>
    <m:data xmlns:m='http://example.org/stuff'>
      <m:photo>/aWKKapGGyQ=</m:photo>
      <m:sig>Faa7vROi2VQ=</m:sig>
    </m:data>
  </soap:Body>
</soap:Envelope>
```

XOP-encoded SOAP message

Example 193 shows an example of a XOP-encoded (that is, MTOM) SOAP message, which you would obtain when MTOM is enabled on the client side (see “Enabling MTOM” on page 343). Contrast this with the plain example from Example 192.

Example 193:*XOP-Encoded SOAP Message*

```
MIME-Version: 1.0
Content-Type: Multipart/Related;boundary=MIME_boundary;
    type="application/xop+xml";
    start="<mymessage.xml@example.org>";
    startinfo="application/soap+xml; action=\"ProcessData\""
Content-Description: A SOAP message with my pic and sig in it

--MIME_boundary
Content-Type: application/xop+xml;
    charset=UTF-8;
    type="application/soap+xml; action=\"ProcessData\""
Content-Transfer-Encoding: 8bit
Content-ID: <mymessage.xml@example.org>

<soap:Envelope
    xmlns:soap='http://www.w3.org/2003/05/soap-envelope'>
  <soap:Body>
    <m:data xmlns:m='http://example.org/stuff'>
      <m:photo><xop:Include
    xmlns:xop='http://www.w3.org/2004/08/xop/include'
    href='cid:http://example.org/me.png'/></m:photo>
      <m:sig><xop:Include
    xmlns:xop='http://www.w3.org/2004/08/xop/include'
    href='cid:http://example.org/my.hsh'/></m:sig>
    </m:data>
  </soap:Body>
</soap:Envelope>
```

Example 193: *XOP-Encoded SOAP Message*

```
--MIME_boundary
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary
Content-ID: <http://example.org/me.png>

// binary octets for png

--MIME_boundary
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary
Content-ID: <http://example.org/my.hsh>

// binary octets for signature

--MIME_boundary--
```

Where, in this case, the SOAP message is encoded as a MIME multipart/related message, including a MIME header and three parts:

1. The first part contains the SOAP message itself. The binary content is no longer embedded directly in the `m:photo` and `m:sig` elements, however. The `m:photo` element contains a `xop:Include` element that references the second message part, using the content ID, `http://example.org/me.png`. The `m:sig` element contains a `xop:Include` element that references the third message part, using the content ID, `http://example.org/my.hsh`.
2. The second part contains the raw binary content of the `m:photo` element. That is, the binary content is *not* base-64 encoded. This enables the binary content to be sent more efficiently and compactly.

   The content of this part is labelled by the `http://example.org/me.png` content ID and the MIME content type is set to `application/octet-stream` by default.
3. The third part contains the raw binary content of the `m:sig` element.

   The content of this part is labelled by the `http://example.org/my.hsh` content ID and the MIME content type is set to `application/octet-stream` by default.

Specifying the MIME Content Type

When binary data is sent in an attachment, it is usual to declare the data format, using a Multipurpose Internet Mail Extensions (MIME) content type descriptor. In the context of MTOM, the XMIME specification describes how to declare the *MIME content type* of data transmitted in a multi-part MTOM message.

The advantage of declaring the MIME content type is that servers can optionally implement MIME content handlers to optimize the processing of the binary data.

xmime:base64Binary with xmime:contentType attribute

The XMIME schema defines the type, `xmime:base64Binary`, which includes the `xmime:contentType` attribute. Example 194 shows how to use the `xmime:base64Binary` type in a WSDL file.

Example 194: *XMIME Base-64 Type with xmime:contentType Attribute*

```
<wsdl:types>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:tns="http://example.og/stuff"
          xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
          targetNamespace="http://example.og/stuff">

<xs:import namespace="http://www.w3.org/2005/05/xmlmime"
     schemaLocation="http://www.w3.org/2005/05/xmlmime"/>
    <xs:element name="data">
      <xs:complexType>
        <xs:sequence>
         <xs:element name="photo" type="xmime:base64Binary"/>
         <xs:element name="sig" type="xmime:base64Binary"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    ...
  </xs:schema>
</wsdl:types>
```

xmime:hexBinary with xmime:contentType attribute

The XMIME schema defines the type, `xmime:hexBinary`, which includes the `xmime:contentType` attribute. Use this type in place of `xs:hexBinary`, if you want to be able to specify the MIME content type of the binary data.

C++ mapping

The WSDL-to-C++ compiler treats the custom binary types specially:

- The `xmime:base64Binary` type and any custom `xs:base64Binary` type that provides the `xmime:contentType` attribute are mapped to the `IT_Bus::XMimeBase64Binary` C++ type.
- The `xmime:hexBinary` type and any custom `xs:hexBinary` type that provides the `xmime:contentType` attribute are mapped to the `IT_Bus::XMimeHexBinary` C++ type.

Setting the MIME content type

You can set the MIME content type on the client side, using the `set_content_type()` member function, as shown in Example 195.

Example 195:*Setting the MIME Content Type*

```
// C++
IT_Bus::XMimeBase64Binary data1(origin_data);
data1.set_content_type("image/png");
```

Plain SOAP message

Example 196 shows an example of a plain, *non-MTOM* SOAP message containing base-64 binary data, where the binary elements include the MIME content type attribute. Both the `m:photo` element and the `m:sig` element include an `xmlmime:contentType` setting.

Example 196:*Plain SOAP Message with MIME Content Type*

```
<soap:Envelope
    xmlns:soap='http://www.w3.org/2003/05/soap-envelope'
    xmlns:xmlmime='http://www.w3.org/2004/11/xmlmime'>
  <soap:Body>
    <m:data xmlns:m='http://example.org/stuff'>
<m:photo xmlmime:contentType='image/png'>/aWKKapGGyQ=</m:photo>
<m:sig xmlmime:contentType='application/pkcs7-signature'>Faa7vROi2VQ=</m:sig>

    </m:data>
  </soap:Body>
</soap:Envelope>
```

XOP-encoded SOAP message

Example 197 shows the equivalent XOP-encoded SOAP message which you would obtain when MTOM is enabled. Contrast this with the plain example from Example 196.

Example 197:*XOP-Encoded SOAP Message with MIME Content Type*

```
MIME-Version: 1.0
Content-Type: Multipart/Related;boundary=MIME_boundary;
    type="application/xop+xml";
    start="<mymessage.xml@example.org>";
    startinfo="application/soap+xml; action=\"ProcessData\""
Content-Description: A SOAP message with my pic and sig in it

--MIME_boundary
Content-Type: application/xop+xml;
    charset=UTF-8;
    type="application/soap+xml; action=\"ProcessData\""
Content-Transfer-Encoding: 8bit
Content-ID: <mymessage.xml@example.org>

<soap:Envelope
```

Example 197: *XOP-Encoded SOAP Message with MIME Content Type*

```
    xmlns:soap='http://www.w3.org/2003/05/soap-envelope'
    xmlns:xmlmime='http://www.w3.org/2004/11/xmlmime'>
  <soap:Body>
    <m:data xmlns:m='http://example.org/stuff'>
      <m:photo
  xmlmime:contentType='image/png'><xop:Include
    xmlns:xop='http://www.w3.org/2004/08/xop/include'
    href='cid:http://example.org/me.png'/></m:photo>
      <m:sig
  xmlmime:contentType='application/pkcs7-signature'><xop:Include
    xmlns:xop='http://www.w3.org/2004/08/xop/include'
    href='cid:http://example.org/my.hsh'/></m:sig>
    </m:data>
  </soap:Body>
</soap:Envelope>

--MIME_boundary
Content-Type: image/png
Content-Transfer-Encoding: binary
Content-ID: <http://example.org/me.png>

// binary octets for png

--MIME_boundary
Content-Type: application/pkcs7-signature
Content-Transfer-Encoding: binary
Content-ID: <http://example.org/my.hsh>

// binary octets for signature

--MIME_boundary--
```

Restricting the MIME Content Type

XMIME allows you to annotate your element definitions to restrict the range of MIME types that can be sent in the binary data type. Currently, Artix does *not* enforce these restrictions, but nevertheless allows you to set the relevant attribute (that is, `xmime:expectedContentType`) for the sake of interoperability.

Annotating elements with xmime:expectedContentType

To declare the MIME content type (or types) that an element is expected to contain, set the `xmime:expectedContentType` attribute on the element definition in the schema.

For example, to specify that the `photo` element can contain only `image/png` content and the `sig` element can contain only `application/pkcs7-signature` content, define the elements as shown in Example 198.

Example 198: *Elements with xmime:expectedContentType Annotation*

```
<wsdl:types>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:tns="http://example.og/stuff"
          xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
          targetNamespace="http://example.og/stuff">

<xs:import namespace="http://www.w3.org/2005/05/xmlmime"
     schemaLocation="http://www.w3.org/2005/05/xmlmime"/>
    <xs:element name="data">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="photo"
              type="xmime:base64Binary"
      xmime:expectedContentType="image/png"/>
          <xs:element name="sig"
              type="xmime:base64Binary"
       xmime:expectedContentType="application/pkcs7-signature"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    ...
  </xs:schema>
</wsdl:types>
```

Note the contrasting roles played by the `xmime:contentType` attribute and the `xmime:expectedContentType` attribute:

- The `xmime:contentType` attribute is set on the `xmime:base64Binary` element. Its value is defined by the client, at run time.
- The `xmime:expectedContentType` attribute is set on the `xs:element` element. Its value is defined in the WSDL contract.

C++ mapping

Under normal circumstances, the `xmime:expectedContentType` setting is applied to an element whose type provides an `xmime:contentType` attribute. Hence, when this element's type is mapped to C++, it is mapped to `IT_Bus::XMimeBase64Binary` or as `IT_Bus::XMimeHexBinary`, as described in "C++ mapping" on page 346.

A special case arises, however, when the `xmime:expectedContentType` setting is applied to an element whose type does *not* provide an `xmime:contentType` attribute (that is, the `xmime:expectedContentType` is used on its own). This case only makes sense, if the `xmime:expectedContentType` setting specifies a *single* expected content type, in which case the MIME content type is implicit.

Syntax of expected content type

The `xmime:expectedContentType` attribute is normally set to a comma-separated list of MIME types. For example, to restrict the content type of the `photo` element to `image/jpeg` or `image/png`, you could define it as follows:

```
<xs:element name="photo"
    type="xmime:base64Binary"
    xmime:expectedContentType="image/jpeg, image/png"/>
```

For full details of the expected content type syntax, see section 14.1 of RFC-2616. Although supported by RFC-2616, the XMIME specification recommends that you do *not* use wildcard expressions (for example, `image/*`) in your expected content type expressions. Wildcard expressions could potentially lead to interoperability problems.

Note: The value of the `xmime:expectedContentType` attribute provides a hint to the Artix WSDL-to-Java compiler to generate the special MIME binary types. Otherwise, this attribute has no effect. In particular, the implied restriction on the value of the `xmime:contentType` attribute is *not* enforced.

Wildcarding Types

The XML schema wildcarding types enable you to define XML types with loosely defined characteristics. The following features of an XML element can be wildcarded:

- *Attribute wildcard,* `xsd:anyAttribute`—matches any attribute. For example, you could use an attribute wildcard to define an element that can have arbitrary attributes.
- *URI wildcard,* `xsd:anyURI`—matches any URI. For example, you could specify `xsd:anyURI` as the type of an attribute that can be initialized with a URI.
- *Contents wildcard,* `xsd:anyType`—matches any XML type for the element contents. For example, you can specify `type="xsd:anyType"` in an element definition to indicate that the element contents may be of any type.
- *Element wildcard,* `xsd:any`—matches any XML element. For example, you could use an element wildcard to define a complex type containing an arbitrary element or elements.

This section contains the following types:

anyAttribute Type

If you include the `<xsd:anyAttribute/>` tag in a complex type definition, it enables you to associate arbitrary attributes with that complex type. The `anyAttribute` element matches any number of attributes by default.

anyAttribute syntax

To declare an `<xsd:anyAttribute>` attribute wildcard, use the following syntax:

```
<xsd:anyAttribute
  id="ID"
  namespace="NamespaceList"
  processContents="(lax | skip | strict)" />
```

Note: Artix does not enforce the `id`, `namespace`, or `processContents` settings that appear in the `anyAttribute` definition.

Namespace constraint

You can use a namespace constraint to restrict the matching attributes to belong to a particular namespace or namespaces. The following values can be specified in the `namespace` attribute:

`##any`	(Default) Matches attributes in any namespace.
`##local`	Matches an unqualified attribute (no namespace prefix appearing in the attribute name).
`##targetNamespace`	Matches attributes in the current `targetNamespace`.
`##other`	Matches attributes in any namespace apart from the current `targetNamespace`.
Namespace	Matches attributes in the literal *Namespace*.
List of namespaces	A space-separated list of namespaces. The list can include literal namespaces, `##targetNamespace`, or `##local`.

Process contents

The `processContents` attribute is an instruction to the XML parser indicating how strictly it should check the syntax of the matched attributes. Sometimes it can be useful to disable syntax checking, because the XML schema for the matched attributes might not be readily available. The `processContents` attribute can have one of the following values:

`strict`	(Default) A schema definition for the attribute must be available and the attribute must conform to this definition.

`lax`	The parser checks the attribute only if a schema definition is available.
`skip`	No checking is done against a schema.

WSDL any example

Example 199 shows the definition of a complex type, `SeqAnyAttributes`, which can include arbitrary attributes.

Example 199: *Definition of a Sequence with Any Attributes*

```
<schema targetNamespace="..."
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <complexType name="SeqAnyAttributes">
        <sequence>
            <element name="stringEl" type="string"/>
            <element name="intEl" type="int"/>
        </sequence>
        <attribute name="stringAt" type="string"/>
       <anyAttribute/>
    </complexType>
    ...
</schema>
```

C++ mapping

When the `SeqAnyAttributes` type maps to C++, the presence of the `<anyAttribute/>` tag prompts the WSDL-to-C++ compiler to generate additional member functions, as shown in Example 200.

Example 200: *C++ Mapping of a Sequence with Any Attributes*

```
// C++
class SeqAnyAttributes : public IT_Bus::SequenceComplexType
{
  public:
    ...
    IT_Bus::QNameHashMap<IT_Bus::String> &
    getotherAttributes();

    const IT_Bus::QNameHashMap<IT_Bus::String> &
    getotherAttributes() const;

    void setotherAttributes(
        const IT_Bus::QNameHashMap<IT_Bus::String> & val
    );
};
```

The additional attributes are accessible in the form of an `IT_Bus::QNameHashMap<IT_Bus::String>` instance, which is a hash map that associates the name of each attribute with a string value. You can use an attribute's QName to access its string value.

IT_Bus::QNameHashMap template class

Example 201 shows how the IT_Bus::QNameHashMap template class is defined in terms of the proprietary IT_HashMap template class. This definition states essentially that the IT_Bus::QName type is used as the hash key.

Example 201:*IT_Bus::QNameHashMap Template Class*

```
// C++
#include <it_dsa/hash_map.h>

namespace IT_Bus
{
    template <class T>
    class QNameHashMap
      : public IT_HashMap<QName, T, QNameHash, QNameEq>
    {
    };
};
```

The IT_HashMap template class is closely modelled on the std::map class from the C++ Standard Template Library. For details of the functions and operations provided by the IT_HashMap class, see "IT_HashMap Template Class" on page 412.

Example

Example 202 shows you how to initialize an instance of the SeqAnyAttributes type defined in Example 199 on page 352. This example uses the anyAttribute mechanism to set two additional attributes: boolAt, an attribute with a boolean value, and floatAt, an attribute with a float value. The additional attributes both belong to the http://test.iona.com namespace.

Example 202:*C++ Setting Any Attributes*

```
// C++
SeqAnyAttributes x;

x.setstringEl("Hello");
x.setintEl(1000);
x.setstringAt("Hello Attribute");

IT_Bus::QNameHashMap<IT_Bus::String> attMap;
IT_Bus::QName at1_qname("", "boolAt", "http://test.iona.com/");
IT_Bus::QName at2_qname("", "floatAt", "http://test.iona.com/");
attMap.insert(
    IT_Bus::QNameHashMap<IT_Bus::String>::value_type(
        at1_qname,
        "true"
    )
);
attMap.insert(
    IT_Bus::QNameHashMap<IT_Bus::String>::value_type(
        at2_qname,
        "3.14"
```

Example 202:*C++ Setting Any Attributes*

```
    )
);
x.setotherAttributes(attMap);
```

anyURI Type

You can specify the `xsd:anyURI` type for any data that is intended to be used as a URI.

anyURI syntax

The `xsd:anyURI` type can be used to define an attribute that holds a URI value or an element that contains a URI value.

To define an attribute with a URI value, use the following syntax:

```
<attribute name="AttrName" type="xsd:anyURI"/>
```

To define an element with URI content, use the following syntax.

```
<element name="ElemName" type="xsd:anyURI"/>
```

C++ mapping

Example 203 shows the most important member functions from the `IT_Bus::AnyURI` class, which is the C++ mapping of `xsd:anyURI`.

Example 203:*The IT_Bus::AnyURI Class*

```
// C++
namespace IT_Bus
{
    class IT_AFC_API AnyURI : public AnySimpleType
    {
      public:
        ...
        AnyURI() IT_THROW_DECL(());
        AnyURI(
            const String & uri
        ) IT_THROW_DECL((IT_Bus::Exception));
        ...
        void set_uri(
            const String & uri
        ) IT_THROW_DECL((IT_Bus::Exception));
        const String& get_uri() const IT_THROW_DECL(());

        static bool is_valid_uri(
            const String & uri
        ) IT_THROW_DECL(());
        ...
    };

    bool operator==(const AnyURI& lhs, const AnyURI& rhs) const;
    bool operator!=(const AnyURI& lhs, const AnyURI& rhs) const;
};
```

If you attempt to set the URI to an invalid value, using either the AnyURI constructor or the set_uri() function, a system exception is thrown.

WSDL example

Example 204 shows an example of a WSDL type, DocReference, that includes an attribute of xsd:anyURI type.

Example 204:*Definition of an Attribute Using an anyURI*

```
<schema targetNamespace="..."
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <complexType name="DocReference">
        <attribute name="doc_type" type="xsd:string"/>
        <attribute name="location" type="xsd:anyURI"/>
    </complexType>
    ...
</schema>
```

Example

The following example code shows how to create an instance of the DocReference type defined in the preceding Example 204. The location attribute is initialized with a URI value.

```
// C++
DocReference dr;

dr.setdoc_type("PDF");
dr.setlocation(
    new IT_Bus::AnyURI("http://www.iona.com/docs/dummy.pdf")
);
```

anyType Type

In an XML schema, the xsd:anyType is the base type from which other simple and complex types are derived. Hence, an element declared to be of xsd:anyType type can contain any XML type.

Note: The xsd:anyType is currently supported only by the CORBA, SOAP and XML bindings. Certain bindings—for example, Fixed, Tagged, TibMsg, and FML—do not support the use of xsd:anyType because they lack a corresponding construct.

Prerequisite for using anyType

A prerequisite for using the xsd:anyType is that your application must be built with the *WSDLFileName*_wsdlTypesFactory.cxx source file. This file is generated automatically by the WSDL-to-C++ compiler utility.

anyType syntax

To declare an `xsd:anyType` element, use the following syntax:

```
<element name="ElementName" [type="xsd:anyType"]>
```

The attribute setting, `type="xsd:anyType"`, is optional. If the `type` attribute is missing, the XML schema assumes that the element is of `xsd:anyType` by default.

C++ mapping

The WSDL-to-C++ compiler maps the `xsd:anyType` type to the `IT_Bus::AnyHolder` class in C++.

The `IT_Bus::AnyHolder` class provides member functions to insert and extract data values, as follows:

- Inserting and extracting atomic types.
- Inserting and extracting user-defined types.

Note: It is currently not possible to nest an `IT_Bus::AnyHolder` instance directly inside another `IT_Bus::AnyHolder` instance.

Inserting and extracting atomic types

To insert and extract atomic types to and from an `IT_Bus::AnyHolder`, use the member functions of the following form:

```
void set_AtomicTypeFunc(const AtomicTypeName&);
AtomicTypeName& get_AtomicTypeFunc();
const AtomicTypeName& get_AtomicTypeFunc();
```

For a complete list of the functions for the basic atomic types, see "AnyHolder API" on page 358.

For example, you can insert and extract an `xsd:short` integer to and from an `IT_Bus::AnyHolder` as follows:

```
// C++
// Insert an xsd:short value into an xsd:anyType.
IT_Bus::AnyHolder aH;
aH.set_short(1234);
...
// Extract an xsd:short value from an xsd:anyType.
IT_Bus::Short sh = aH.get_short();
```

Inserting and extracting user-defined types

To insert and extract user-defined types from an IT_Bus::AnyHolder, use the following functions:

```
void                    set_any_type(const IT_Bus::AnyType &);
IT_Bus::AnyType&        get_any_type();
const IT_Bus::AnyType&  get_any_type();
```

Note that all user-defined types inherit from IT_Bus::AnyType. There are no type-specific insertion or extraction functions generated for user-defined types.

Memory management for these functions is handled as follows:

- The set_any_type() function copies the inserted data.
- The get_any_type() functions do not copy the return value, rather they return either a writable (non-const) or read-only (const) reference to the data inside the IT_Bus::AnyHolder.

For example, given a user-defined sequence type, SequenceType (see the declaration in Example 149 on page 310), you can insert a SequenceType instance into an IT_Bus::AnyHolder as follows:

```
// C++
// Create an instance of SequenceType type.
SequenceType seq;
seq.setvarFloat(3.14);
seq.setvarInt(1234);
seq.setvarString("This is a sample SequenceType.");

// Insert the SequenceType value into an xsd:anyType.
IT_Bus::AnyHolder aH;
aH.set_any_type(seq);
```

To extract the SequenceType instance from the IT_Bus::AnyHolder, you need to perform a C++ dynamic cast:

```
// C++
...
// Extract the SequenceType value from the IT_Bus::AnyHolder.
IT_Bus::AnyType& base_extract = aH.get_any_type();

// Cast the extracted value to the appropriate type:
SequenceType& seq_extract
                  = dynamic_cast<SequenceType&>(base_extract);
```

Accessing the type information

You can find out what type of data is contained in an IT_Bus::AnyHolder instance by calling the following member function:

```
const IT_Bus::QName & get_type() const;
```

Type information is set whenever an IT_Bus::AnyHolder instance is initialized. For example, if you initialize an IT_Bus::AnyHolder by calling set_boolean(), the type is set to be xsd:boolean. If you call set_any_type() with an argument of SequenceType, the type would be set to xsd1:SequenceType.

Note: Because the XML representation of xsd:anyType is not self-describing, some type information could be lost when an anyType is sent across the wire. In the case of a CORBA binding, however, there is no loss of type information, because CORBA anys are fully self-describing.

AnyHolder API

Example 205 shows the public API from the IT_Bus::AnyHolder class, including all of the function for inserting and extracting data values.

Example 205:*The IT_Bus::AnyHolder Class*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API AnyHolder : public AnyType
    {
      public:
        AnyHolder();
        virtual ~AnyHolder() ;
        ...
        virtual const QName & get_type() const ;
        ...
        //Set Methods
        void set_boolean(const IT_Bus::Boolean &);
        void set_byte(const IT_Bus::Byte &);
        void set_short(const IT_Bus::Short &);
        void set_int(const IT_Bus::Int &);
        void set_long(const IT_Bus::Long &);
        void set_string(const IT_Bus::String &);
        void set_float(const IT_Bus::Float &);
        void set_double(const IT_Bus::Double &);
        void set_ubyte(const IT_Bus::UByte &);
        void set_ushort(const IT_Bus::UShort &);
        void set_uint(const IT_Bus::UInt &);
        void set_ulong(const IT_Bus::ULong &);
        void set_decimal(const IT_Bus::Decimal &);

        void set_any_type(const AnyType&);

        //GET METHODS
        IT_Bus::Boolean & get_boolean();
        IT_Bus::Byte & get_byte();
        IT_Bus::Short & get_short();
        IT_Bus::Int & get_int();
        IT_Bus::Long & get_long();
        IT_Bus::String & get_string();
        IT_Bus::Float & get_float();
        IT_Bus::Double & get_double();
```

Example 205: *The IT_Bus::AnyHolder Class*

```
        IT_Bus::UByte & get_ubyte() ;
        IT_Bus::UShort & set_ushort();
        IT_Bus::UInt & get_uint();
        IT_Bus::ULong & set_ulong();
        IT_Bus::Decimal & get_decimal();

        AnyType& get_any_type();

        //CONST GET METHODS
        const IT_Bus::Boolean & get_boolean() const;
        const IT_Bus::Byte & get_byte() const;
        const IT_Bus::Short & get_short() const;
        const IT_Bus::Int & get_int() const;
        const IT_Bus::Long & get_long() const;
        const IT_Bus::String & get_string() const;
        const IT_Bus::Float & get_float() const;
        const IT_Bus::Double & get_double() const;
        const IT_Bus::UByte & get_ubyte() const;
        const IT_Bus::UShort & get_ushort() const;
        const IT_Bus::UInt & get_uint() const;
        const IT_Bus::ULong & get_ulong() const;
        const IT_Bus::Decimal & get_decimal() const;

        const AnyType& get_any_type() const;
        ...
    };
};
```

any Type

In an XML schema, the `xsd:any` is a wildcard element that matches any element (or multiple elements, if occurrence constraints are set), subject to certain constraints.

any syntax

To declare an `xsd:any` element, use the following syntax:

```
<xsd:any
  minOccurs="LowerBound"
  maxOccurs="UpperBound"
  namespace="NamespaceList"
  processContents="(lax | skip | strict)" />
```

Occurrence constraints

You can use occurrence constraints to specify how many elements can be matched by the `xsd:any` element wildcard:

- `minOccurs` specifies the minimum number of elements to match (default `1`).
- `maxOccurs` specifies the maximum number of elements to match (default `1`).

For more details about implementing `any`s with occurrence constraints, see “Any Occurrence Constraints” on page 375.

Target namespace

An `xsd:any` element is implicitly associated with a particular target namespace (specified by the `targetNamespace` attribute in one of the elements enclosing the `<xsd:any>` definition).

Namespace constraint

You can use a namespace constraint to restrict the matching elements to belong to a particular namespace or namespaces. The following values can be specified in the `namespace` attribute:

`##any`	(Default) Matches elements in any namespace, including unqualified elements.
`##local`	Matches an unqualified element (no namespace prefix appearing in the element name).
`##targetNamespace`	Matches elements in the current `targetNamespace`.
`##other`	Matches elements in any namespace apart from the current `targetNamespace`.
Namespace	Matches elements in the literal *Namespace*.
List of namespaces	A space-separated list of namespaces. The list can include literal namespaces, `##targetNamespace`, or `##local`.

Process contents

The `processContents` attribute is an instruction to the XML parser indicating how strictly it should check the syntax of the matched elements. Sometimes it can be useful to disable syntax checking, because the XML schema for the matched elements might not be readily available. The `processContents` attribute can have one of the following values:

`strict`	(Default) A schema definition for the element type must be available and the element must conform to this definition.
`lax`	The parser checks only those parts of the element for which a schema definition is available.
`skip`	No checking is done against a schema; the element must simply be well-formed XML.

WSDL any example

Example 206 shows the definition of a complex type, `SequenceAny`, which can contain a single element tag from the local schema. That is, the `<any>` tag is constrained to match only the tags belonging to the local namespace.

Example 206: *Definition of a Sequence with an Any Element*

```
<schema targetNamespace="..."
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <complexType name="SequenceAny">
        <sequence>
            <any namespace="##local"
                 processContents="skip"/>
        </sequence>
    </complexType>
    ...
</schema>
```

C++ mapping

The XML `SequenceAny` type defined in Example 206 on page 361 maps to the C++ `SequenceAny` class shown in Example 207. The most important functions in `SequenceAny` are the `getany()` and `setany()` members, which access or modify the any element in the sequence.

Example 207: *C++ Mapping of a Sequence with an Any Element*

```
// C++
class SequenceAny : public IT_Bus::SequenceComplexType
{
  public:
    ...
    SequenceAny();
    SequenceAny(const SequenceAny & copy);
    virtual ~SequenceAny();

    IT_Bus::AnyType & copy(const IT_Bus::AnyType & rhs);
    SequenceAny & operator=(const SequenceAny & rhs);

    IT_Bus::Any &       getany();
    const IT_Bus::Any & getany() const;
    void setany(const IT_Bus::Any & val);
    ...
};
```

Example XML element

Example 208 shows the definition of a sample foo element, which can be inserted in place of an any element.

Example 208:*Definition of fooType Type and foo Element*

```
// C++
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://schemas.iona.com/test"
        xmlns:tns="http://schemas.iona.com/test"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">
    <xs:complexType name="fooType">
        <xs:simpleContent>
            <xs:extension base="xs:string"/>
        </xs:simpleContent>
        <xs:attribute name="bar" type="xs:string"/>
    </xs:complexType>
    <xs:element name="foo" type="tns:fooType"/>
</xs:schema>
```

Example

There are two alternative approaches to initializing an IT_Bus::Any value.

The first approach to initializing IT_Bus::Any is to call the set_any_type() function, as shown in the following example:

```
// C++
fooType foo_element;
foo_element.setvalue("Hello World!");
foo_element.setbar("bar attribute value");

IT_Bus::QName
   element_name("","foo","http://schemas.iona.com/test");

SequenceAny seq_any;
seq_any.getany().set_any_type(foo_element, element_name);
```

The second approach to initializing IT_Bus::Any is to call the set_string_data() function, as shown in the following example:

```
// C++
SequenceAny seq_any;
seq_any.getany().set_string_data(
    "<foo bar=\"bar attribute value\">Hello World!</foo>"
);
```

Any API

Example 209 shows the public API from the `IT_Bus::Any` class.

Example 209: *The IT_Bus::Any Class*

```
// C++
namespace IT_Bus
{
    typedef IT_Vector<String> NamespaceConstraints;

    class IT_AFC_API Any : public AnyType
    {
      public :
        Any();

        Any(const char*                  process_contents,
            const NamespaceConstraints&  namespace_constraints,
            const char*                  any_namespace
        );
        ...
        // Set the any element's attributes.
        void set_process_contents(const String& pc);
        void set_namespace_constraints(
            const NamespaceConstraints& ns
        );
        void set_any_namespace(const String& ns);

        // Get the any element's attributes.
        String& get_process_contents() const;
        const NamespaceConstraints&
        get_namespace_constraints() const;
        String& get_any_namespace() const;

       // Set the any's contents.
        void set_boolean(
            const Boolean& value,
            const QName&   element_name
        );
        void set_byte(
            const Byte&    value,
            const QName&   element_name
        );
        void set_short(
            const Short&   value,
            const QName&   element_name
        );
        void set_int(
            const Int&     value,
            const QName&   element_name
        );
        void set_long(
            const Long&    value,
            const QName&   element_name
        );
        void set_string(
            const String& value,
            const QName&   element_name
        );
```

Example 209: *The IT_Bus::Any Class*

```
        void set_float(
            const Float&   value,
            const QName&   element_name
        );
        void set_double(
            const Double&  value,
            const QName&   element_name
        );
        void set_ubyte(
            const UByte&   value,
            const QName&   element_name
        ) ;
        void set_ushort(
            const UShort&  value,
            const QName&   element_name
        );
        void set_uint(
            const UInt&    value,
            const QName&   element_name
        );
        void set_ulong(
            const ULong&   value,
            const QName&   element_name
        );
        void set_decimal(
            const Decimal&  value,
            const QName&    element_name
        );

        void set_any_type(
            const AnyType&  value,
            const QName&    element_name
        );

        // Get the type of the any's contents.
        // (returns QName::EMPTY_QNAME if empty)
        const QName& get_type() const;

       // Get the any's contents.
        QName get_element_name() const;

        Boolean get_boolean() const;
        Byte    get_byte() const;
        Short   get_short() const;
        Int     get_int() const;
        Long    get_long() const;
        String  get_string() const;
        Float   get_float() const;
        Double  get_double() const;
        UByte   get_ubyte() const;
        UShort  get_ushort() const;
        UInt    get_uint() const;
        ULong   get_ulong() const;
        Decimal get_decimal() const;

        const AnyType* get_any_type() const;
```

Example 209:*The IT_Bus::Any Class*

```
        // Set the any's contents as an XML string
         // (the element_name parameter defaults to the
         //  element name in the XML string).
         void set_string_data(
             const String&  value,
             const QName&   element_name = QName::EMPTY_QNAME
         );

        // Get the any's contents as an XML string.
         String get_string_data() const;

        // Validation functions.
         virtual bool validate_contents() const;
         virtual bool validate_namespace() const;
    };
};
```

Accessing namespace constraints

The following `IT_Bus::Any` member functions are relevant to namespace constraints:

```
// C++
const IT_Bus::String& get_any_namespace() const;

const IT_Bus::NamespaceConstraints&
get_namespace_constraints() const;
```

Given an `IT_Bus::Any` instance, `sampleAny`, you can access its namespace constraints as follows:

```
// C++
sampleAny = ... ; // Initialize IT_Bus::Any
cout << "any's target namespace = "
     << sampleAny.get_any_namespace() << endl;

const IT_Bus::NamespaceConstraints& constraints =
   sampleAny.get_namespace_constraints();
cout << "any's namespace constraints =" << endl;
for (size_t k; k < constraints.size(); k++) {
    cout << "\t" << constraints[k] << endl;
}
```

Accessing process contents

The following `IT_Bus::Any` member function returns the `processContents` attribute value:

```
const IT_Bus::String& get_process_contents() const;
```

This function returns one of the following strings: `lax`, `skip`, or `strict`.

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.

This section contains the following subsections:

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 334.

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.

Element lists

Lists of elements appearing within a sequence complex type are represented in C++ by the `IT_Bus::ElementListT` template, which inherits from `IT_Vector` (see "IT_Vector Template Class" on page 408).

In addition to the standard member functions and operators defined by `IT_Vector`, the element list types support the following member functions:

```
// C++
size_t get_min_occurs() const;
void   set_min_occurs(size_t min_occurs)
```

```
size_t get_max_occurs() const;
void   set_max_occurs(size_t max_occurs)

void   set_size(size_t new_size);

size_t get_size() const;

const QName & get_item_name() const;
void set_item_name(const QName& item_name)
```

Element list constructor

The following constructor can be used to create a new `ElementListT` instance:

```
ElementListT(
    const size_t min_occurs = 0,
    const size_t max_occurs = 1,
    const size_t list_size = 0,
    const QName& item_name = QName::EMPTY_QNAME
);
```

It is recommended that you call only the form of constructor with defaulted arguments (the element list size can be specified subsequently by calling `set_size()`). For example, a new element list of integers could be created as follows:

```
IT_Bus::ElementListT<IT_Bus::Int> int_elist;
int_elist.set_size(100);
...
```

When the element list is subsequently passed as a parameter or return value, the stub code takes responsibility for filling in the correct values of `min_occurs`, `max_occurs`, and `item_name`.

WSDL example

Example 210 shows the definition of a sequence type, `SequenceType`, which contains a list of integer elements followed by a list of string elements.

Example 210:*Sequence Type with Element Occurrence Constraints*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >
  <types>
    <schema ... >
      <complexType name="SequenceType">
        <sequence>
          <element name="varInt" type="xsd:int"
                  minOccurs="1" maxOccurs="100"/>
          <element name="varString" type="xsd:string"
                  minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
      </complexType>
    ...
  ...
</definitions>
```

C++ mapping

Example 211 shows an outline of the C++ `SequenceType` class generated from Example 210 on page 368, which defines accessor and modifier functions for the `varInt` and `varString` elements.

Example 211:*Mapping of SequenceType to C++*

```
// C++
class SequenceType : public IT_Bus::SequenceComplexType
{
  public:
  ...
  virtual const IT_Bus::QName &
  get_type() const;

  SequenceType& operator= (const SequenceType& assign);

  const IT_Bus::ElementListT<IT_Bus::Int> & getvarInt() const;

  IT_Bus::ElementListT<IT_Bus::Int> & getvarInt();

  void setvarInt(const IT_Bus::ElementListT<IT_Bus::Int> & val);

  const IT_Bus::ElementListT<IT_Bus::String> & getvarString() const;

  IT_Bus::ElementListT<IT_Bus::String> & getvarString();

  void setvarString(const IT_Bus::ElementListT<IT_Bus::String> & val);

  private:
  ...
};
```

Example

The following code fragment shows how to allocate and initialize an instance of `SequenceType` type containing two `varInt` elements and two `varString` elements:

```
// C++
SequenceType seq;

seq.getvarInt().set_size(2);
seq.getvarInt()[0] = 10;
seq.getvarInt()[1] = 20;
seq.getvarString().set_size(2);
seq.getvarString()[0] = "Zero";
seq.getvarString()[1] = "One";
```

Note how the `set_size()` function and `[]` operator are invoked directly on the member vectors, which are accessed by `getvarInt()` and `getvarString()` respectively. This is more efficient than creating a vector and passing it to `setvarInt()` or `setvarString()`, because it avoids creating unnecessary temporary vectors.

Alternatively, you could assign the member vectors, `seq.getvarInt()` and `seq.getvarString()`, to references of `ElementListT` type and manipulate the references, `v1` and `v2`, instead. This is shown in the following code example:

```
// C++
SequenceType seq;

// Make a shallow copy of the vectors
IT_Bus::ElementListT<IT_Bus::Int>&    v1 = seq.getvarInt();
IT_Bus::ElementListT<IT_Bus::String>& v2 = seq.getvarString();

v1.push_back(10);
v1.push_back(20);
v2.push_back("Zero");
v2.push_back("One");
```

In this example, the vectors are initialized using the `push_back()` stack operation (adds an element to the end of the vector).

References

For more details about vector types see:

- The "IT_Vector Template Class" on page 408.
- The section on C++ ANSI vectors in *The C++ Programming Language*, third edition, by Bjarne Stroustrup.

Sequence Occurrence Constraints

A `sequence` type can also be defined with occurrence constraints, in which case it is defined with the following syntax:

```
<sequence
```

```
  minOccurs="LowerBound"
  maxOccurs="UpperBound">
    ...
</sequence>
```

Note: A sequence with occurrence constraints is currently supported only by the SOAP binding.

WSDL example

Example 212 shows the definition of a sequence type, CultureInfo, with sequence occurrence constraints. The sequence overall can be repeated 0 to 2 times. The Name element within the sequence can also be repeated a variable number of times, from 0 to 1 times.

Example 212:*Sequence Occurrence Constraints*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >
  <types>
    <schema ... >
       <complexType name="CultureInfo">
           <sequence minOccurs="0" maxoccurs="2">
               <element minOccurs="0" maxOccurs="1" name="Name"
                        type="string"/>
               <element minOccurs="1" maxOccurs="1" name="Lcid"
                        type="int"/>
           </sequence>
           <attribute name="varAttrib" type="string"/>
       </complexType>
    ...
  ...
</definitions>
```

C++ mapping

Example 213 shows an outline of the C++ CultureInfo class generated from Example 212 on page 370, which defines accessor and modifier functions for the Name and Lcid elements.

Example 213:*Mapping CultureInfo to C++*

```
// C++
class CultureInfo : public IT_Bus::SequenceComplexType
{
  public:
    static const IT_Bus::QName& get_static_type();

    CultureInfo();
    CultureInfo(const CultureInfo & copy);
    virtual ~CultureInfo();
    ...
    virtual const IT_Bus::QName & get_type() const;

    size_t get_min_occurs() const;
```

Example 213: *Mapping CultureInfo to C++*

```
    size_t get_max_occurs() const;

    void set_size(size_t new_size);
    size_t get_size() const;
    ...
    IT_Bus::ElementListT<IT_Bus::String> &
    getName(size_t seq_index = 0);

    const IT_Bus::ElementListT<IT_Bus::String> &
    getName(size_t seq_index = 0) const;

    void
    setName(
        const IT_Vector<IT_Bus::String> & val,
        size_t seq_index = 0
    );

    IT_Bus::Int       getLcid(size_t seq_index = 0);

    const IT_Bus::Int getLcid(size_t seq_index = 0) const;

    void setLcid(const IT_Bus::Int val, size_t seq_index = 0);
    ...
    IT_Bus::String&       getvarAttrib() const;
    const IT_Bus::String& getvarAttrib();
    void setvarAttrib(const IT_Bus::String& val);
};
```

Member functions

The occurrence constraints on the `sequence` element can be accessed by calling the `get_min_occurs()` and the `get_max_occurs()` member functions.

The number of occurrences of the `sequence` element can be modified and accessed by calling the `set_size()` function and the `get_size()` function, respectively. The default size is `0`; hence, you always need to call `set_size()` to pre-allocate the `sequence` element occurrences.

The functions for getting and setting member elements—for example, `getName()`, `setName()`, `getLcid()`, and `getLcid()`—take an extra final parameter, `seq_index`, that specifies which occurrence is being accessed or modified (the parameter defaults to `0`).

The functions for accessing and modifying an attribute—for example, `getvarAttrib()` and `setvarAttrib()`—do *not* take a `seq_index` parameter. Attributes are always single valued.

Backward compatibility

The mapping to C++ of a sequence type with multiple occurrences is designed to be backward compatible with the default case (`minOccurs="1"`, `maxOccurs="1"`).

For example, it doesn't matter whether the `CultureInfo` type is defined with `minOccurs="1"`, `maxOccurs="1"` or some other value of occurrence constraints; in both cases, the `CultureInfo` XML type maps to a `CultureInfo` C++ class. In the signatures of the element accessors/modifiers, the sequence index defaults to `0`, which is compatible with the default (single occurrence) case.

Note: With non-default occurrence constraints, however, it is necessary to add a line of code to allocate occurrences using `set_size()`, because in this case the default size is `0`.

Example

The following code fragment shows how to allocate and initialize a `CultureInfo` type containing two sequence occurrences, each of which contains one `Name` element and one `Lcid` element:

```
// C++
CultureInfo seq;

// Pre-allocate 2 <sequence> occurrences.
seq.set_size(2);

// First <sequence> occurrence
seq.getName(0).set_size(1);
seq.getName(0)[0] = "First <sequence> occurrence";
seq.setLcid(123, 0);

// Second <sequence> occurrence
seq.getName(1).set_size(1);
seq.getName(1)[0] = "Second <sequence> occurrence";
seq.setLcid(234, 1);

// Set attribute
seq.setvarAttrib("Valid for all <sequence> occurrences.");
```

Notice that the attribute, `varAttrib`, is valid for all occurrences of the `sequence` element. Hence, there is no need for a sequence index in the call to `setvarAttrib()`.

Choice Occurrence Constraints

A `choice` type can also be defined with occurrence constraints, in which case it is defined with the following syntax:

```
<choice
  minOccurs="LowerBound"
  maxOccurs="UpperBound">
    ...
</choice>
```

Note: A `choice` with occurrence constraints is currently supported only by the SOAP binding.

WSDL example

Example 214 shows the definition of a choice type, `ClubEvent`, with choice occurrence constraints. The choice type overall can be repeated 0 to unbounded times.

Example 214:*Choice Occurrence Constraints*

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"

   targetNamespace="http://schemas.iona.com/choice_example">

    <complexType name="ClubEvent">
        <choice minOccurs="0" maxOccurs="unbounded">
            <element name="MemberName" type="xsd:string"/>
            <element name="GuestName" type="xsd:string"/>
        </choice>
    </complexType>

</schema>
```

C++ mapping

Example 215 shows an outline of the C++ `ClubEvent` class generated from Example 214 on page 373, which defines accessor and modifier functions for the `MemberName` and `GuestName` elements.

Example 215:*Mapping ClubEvent to C++*

```
// C++
class ClubEvent : public IT_Bus::ChoiceComplexType
{
  public:
    static const IT_Bus::QName&    get_static_type();

    ClubEvent();
    ClubEvent(const ClubEvent & copy);
    ClubEvent(size_t size);

    virtual ~ClubEvent();

    ...

    size_t get_min_occurs() const { ... }
    size_t get_max_occurs() const { ... }

    size_t get_size() const { ... }
    void   set_size(size_t new_size) { ... }

    ...

    IT_ClubEventChoice::IT_ClubEventChoiceDiscriminator
    get_discriminator(size_t index) const { ... }

    IT_Bus::UInt
    get_discriminator_as_uint(size_t index) const { ... }
```

Example 215: *Mapping ClubEvent to C++*

```
    IT_ClubEventChoice::IT_ClubEventChoiceDiscriminator
    get_discriminator() const { ... }

    IT_Bus::UInt
    get_discriminator_as_uint() const { ... }

    IT_Bus::String &
    getMemberName(size_t seq_index = 0);

    const IT_Bus::String &
    getMemberName(size_t seq_index = 0) const;

    void
    setMemberName(
        const IT_Bus::String & val,
        size_t seq_index = 0
    );

    IT_Bus::String &
    getGuestName(size_t seq_index = 0);

    const IT_Bus::String &
    getGuestName(size_t seq_index = 0) const;

    void
    setGuestName(
        const IT_Bus::String & val,
        size_t seq_index = 0
    );

  private:
    ...
};
```

Member functions

The occurrence constraints on the `choice` element can be accessed by calling the `get_min_occurs()` and the `get_max_occurs()` member functions.

The number of occurrences of the `choice` element can be modified and accessed by calling the `set_size()` function and the `get_size()` function, respectively. The default size is `0`; hence, you always need to call `set_size()` to pre-allocate the `choice` element occurrences.

To access the discriminator value—using `get_discriminator()` or `get_discriminator_as_uint()`—you must supply an `index` parameter to select the relevant occurrence of the choice data.

The functions for getting and setting member elements—for example, getMemberName(), setMemberName(), getGuestName(), and setGuestName()—take an extra final parameter, seq_index, that specifies which occurrence is being accessed or modified (the parameter defaults to 0).

Note: For any attributes are defined on the choice type, the attribute accessors and modifiers do *not* take a seq_index parameter. Attributes are always single valued.

Backward compatibility

The mapping to C++ of a choice type with multiple occurrences is designed to be backward compatible with the default case (minOccurs="1", maxOccurs="1").

For example, it doesn't matter whether the ClubEvent type is defined with minOccurs="1", maxOccurs="1" or some other value of occurrence constraints; in all cases, the ClubEvent XML type maps to a ClubEvent C++ class. In the signatures of the element accessors/modifiers, the sequence index defaults to 0, which is compatible with the default (single occurrence) case.

Note: With non-default occurrence constraints, however, it is necessary to add a line of code to allocate occurrences using set_size(), because in this case the default size is 0.

Example

The following code fragment shows how to allocate and initialize a ClubEvent type containing two choice occurrences:

```
// C++
ClubEvent list;

// Pre-allocate 2 <choice> occurrences.
list.set_size(2);

// First <choice> occurrence
list.setMemberName("Fred Flintstone", 0);

// Second <choice> occurrence
list.setGuestName("Wilma Flintstone", 1);
```

Any Occurrence Constraints

An xsd:any element can also be defined with occurrence constraints, in which case it is defined with the following syntax:

```
<xsd:any
  minOccurs="LowerBound"
  maxOccurs="UpperBound"
  namespace="NamespaceList"
  processContents="(lax | skip | strict)" />
```

WSDL example

Example 216 shows the definition of a complex type, `SequenceAnyList`, which is a sequence containing multiple occurrences of an `<xsd:any>` tag. The `<any>` tag is constrained to match only the tags belonging to the local namespace.

Example 216: *Definition of a Multiply-Occurring Any Element*

```
<schema targetNamespace="..."
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <complexType name="SequenceAnyList">
        <sequence>
            <any namespace="##local"
                    minOccurs="1" maxOccurs="unbounded"
                 processContents="skip"/>
        </sequence>
    </complexType>
    ...
</schema>
```

C++ mapping

The XML `SequenceAnyList` type defined in Example 216 on page 376 maps to the C++ `SequenceAnyList` class shown in Example 217. Because the `SequenceAnyList` type allows multiple occurrences, the `getany()` member function returns `IT_Bus::AnyList` instead of `IT_Bus::Any`, and the `setany()` function takes an `IT_Vector<IT_Bus::Any>` type argument instead of an `IT_Bus::Any` argument.

Example 217: *C++ Mapping of a Multiply-Occurring Any Element*

```
// C++
class SequenceAnyList : public IT_Bus::SequenceComplexType
{
  public:
    ...
    SequenceAnyList();
    SequenceAnyList(const SequenceAnyList & copy);
    virtual ~SequenceAnyList();
    ...
    IT_Bus::AnyList &       getany();
    const IT_Bus::AnyList & getany() const;
    void setany(const IT_Vector<IT_Bus::Any> & val);
    ...
};
```

The IT_Bus::AnyList type

The `IT_Bus::AnyList` class has `IT_Vector<IT_Bus::Any>` as one of its base classes. Hence, the `IT_Bus::AnyList` class is effectively a vector of `IT_Bus::Any` objects. As with any `IT_Vector` type, IT_Bus::AnyList supports a size() function, which gives the number of elements in the list, and a subscripting operator[], which accesses individual elements in the list.

For full details of the IT_Vector<*T*> template, see "IT_Vector Template Class" on page 408.

Example

The following example shows how initialize the SequenceAnyList type with a list of three foo elements (for the schema definition of <foo>, see Example 208 on page 362).

```
// C++
SequenceAnyList seq_any;
IT_Bus::AnyList& any_list = seq_any.getany();
any_list.set_size(3);
any_list[0].set_string_data(
    "<foo bar=\"first bar\">Hello World!</foo>"
);
any_list[1].set_string_data(
    "<foo bar=\"second bar\">Hello World Again!</foo>"
);
any_list[2].set_string_data(
    "<foo bar=\"third bar\">Hello World Yet Again!</foo>"
);
```

IT_Bus::AnyList class

Example 218 shows the public API for the IT_Bus::AnyList class. Typically, you would rarely need to use any of the constructors in this class, because an AnyList object is usually obtained by calling the getany() function on an enclosing type.

Example 218: *The IT_Bus::AnyList Class*

```
// C++
class IT_AFC_API AnyList :
    public TypeListT<Any>
{
  public:
    AnyList(
        const size_t min_occurs,
        const size_t max_occurs,
        const size_t list_size = 0
    );

    AnyList(
        const Any & elem,
        const size_t min_occurs,
        const size_t max_occurs,
        const size_t list_size = 0
    );

    AnyList(
        const size_t min_occurs,
        const size_t max_occurs,
```

Example 218: *The IT_Bus::AnyList Class*

```
        const char*                    process_contents,
        const NamespaceConstraints&  namespace_constraints,
        const char*                    any_tns
    );

    AnyList(
        const size_t min_occurs,
        const size_t max_occurs,
        const size_t list_size,
        const char*                    process_contents,
        const NamespaceConstraints&  namespace_constraints,
        const char*                    any_tns
    );

    AnyList(
        const Any & elem,
        const size_t min_occurs,
        const size_t max_occurs,
        const char*                    process_contents,
        const NamespaceConstraints&  namespace_constraints,
        const char*                    any_tns
    );

    AnyList(
        const Any & elem,
        const size_t min_occurs,
        const size_t max_occurs,
        const size_t list_size,
        const char*                    process_contents,
        const NamespaceConstraints&  namespace_constraints,
        const char*                    any_tns
    );

    virtual ~AnyList() {}

    const String& get_process_contents() const;
    const NamespaceConstraints& get_namespace_constraints() const;
    const String& get_any_namespace() const;

    void set_process_contents(const String &);
    void set_namespace_constraints(const NamespaceConstraints&);
    void set_any_namespace(const String &);

    virtual Kind get_kind() const;
    virtual const QName & get_type() const;

    virtual AnyType& copy(const AnyType & rhs);

    virtual void set_size(size_t new_size);
    ...
};
```

Nillable Types

This section describes how to define and use nillable types; that is, XML elements defined with `xsd:nillable="true"`.

This section contains the following subsections:

- Introduction to Nillable Types
- Nillable Atomic Types
- Nillable User-Defined Types
- Nested Atomic Type Nillable Elements
- Nested User-Defined Nillable Elements
- Nillable Elements of an Array

Introduction to 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`.

On-the-wire format

On the wire, a nil value for an *ElementName* element is represented by the following XML fragment:

```
<ElementName xsi:nil="true"></ElementName>
```

Where the `xsi:` prefix represents the XML schema instance namespace, `http://www.w3.org/2001/XMLSchema-instance`.

C++ API for nillable types

Example 219 shows the public member functions of the `IT_Bus::NillableValueBase` class, which provides the C++ API for nillable types.

Example 219: *C++ API for Nillable Types*

```
// C++
namespace IT_Bus
{
    template <class T>
    class NillableValueBase : public Nillable
    {
```

Example 219: *C++ API for Nillable Types*

```
      public:
        virtual ~NillableValueBase();
        virtual AnyType& operator=(const AnyType& other);

        virtual Boolean is_nil() const;
        virtual void set_nil();
        ...
        virtual const T&
        get() const IT_THROW_DECL((NoDataException));

        virtual T&
        get() IT_THROW_DECL((NoDataException));

        // Set the data value, make is_nil() false.
        virtual void set(const T& data);

        // data != 0 ==> set the data value, make is_nil() false.
        // data == 0 ==> make is_nil() true.
        virtual void set(const T *data);

        // Reset to nil, makes is_nil() true.
        virtual void reset();

      protected:
    ...
};
```

Nillable Atomic Types

This subsection describes how to define and use XML schema nillable atomic types. In C++, every atomic type, *AtomicTypeName*, has a nillable counterpart, *AtomicTypeName*`Nillable`. For example, `IT_Bus::Short` has `IT_Bus::ShortNillable` as its nillable counterpart.

You can modify or access the value of an atomic nillable type, `T`, using the `T.set()` and `T.get()` member functions, respectively. For full details of the API for nillable types see "C++ API for nillable types" on page 379.

Table of nillable atomic types

Table 44 shows how the XML schema atomic types map to C++ when the `xsd:nillable` flag is set to `true`.

Table 44: *Nillable Atomic Types*

Schema Type	Nillable C++ Type
xsd:anyType	*Not supported as nillable*
xsd:boolean	IT_Bus::BooleanNillable
xsd:byte	IT_Bus::ByteNillable
xsd:unsignedByte	IT_Bus::UByteNillable
xsd:short	IT_Bus::ShortNillable

Table 44: *Nillable Atomic Types*

Schema Type	Nillable C++ Type
xsd:unsignedShort	IT_Bus::UShortNillable
xsd:int	IT_Bus::IntNillable
xsd:unsignedInt	IT_Bus::UIntNillable
xsd:long	IT_Bus::LongNillable
xsd:unsignedLong	IT_Bus::ULongNillable
xsd:float	IT_Bus::FloatNillable
xsd:double	IT_Bus::DoubleNillable
xsd:string	IT_Bus::StringNillable
xsd:QName	IT_Bus::QNameNillable
xsd:dateTime	IT_Bus::DateTimeNillable
xsd:date	IT_Bus::DateNillable
xsd:time	IT_Bus::TimeNillable
xsd:gDay	IT_Bus::GDayNillable
xsd:gMonth	IT_Bus::GMonthNillable
xsd:gMonthDay	IT_Bus::GMonthDayNillable
xsd:gYear	IT_Bus::GYearNillable
xsd:gYearMonth	IT_Bus::GYearMonthNillable
xsd:decimal	IT_Bus::DecimalNillable
xsd:integer	IT_Bus::IntegerNillable
xsd:positiveInteger	IT_Bus::PositiveIntegerNillable
xsd:negativeInteger	IT_Bus::NegativeIntegerNillable
xsd:nonPositiveInteger	IT_Bus::NonPositiveIntegerNillable
xsd:nonNegativeInteger	IT_Bus::NonNegativeIntegerNillable
xsd:base64Binary	IT_Bus::BinaryBufferNillable
xsd:hexBinary	IT_Bus::BinaryBufferNillable

WSDL example

Example 220 defines four elements, `test_string_x`, `test_short_y`, `test_int_return`, and `test_float_z`, of nillable atomic type. This example shows how to use the nillable atomic types as the parameters of an operation, `send_receive_nil_part`.

Example 220:*WSDL Example Showing Some Nillable Atomic Types*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BaseService" targetNamespace="http://soapinterop.org/"
```

Example 220: *WSDL Example Showing Some Nillable Atomic Types*

```
...
xmlns:tns="http://soapinterop.org/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsd1="http://soapinterop.org/xsd">
<types>
    <schema targetNamespace="http://soapinterop.org/xsd"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
        ...
       <element name="test_string_x" nillable="true"
               type="xsd:string"/>
       <element name="test_short_y" nillable="true"
               type="xsd:short"/>
       <element name="test_int_return" nillable="true"
               type="xsd:int"/>
       <element name="test_float_z" nillable="true"
               type="xsd:float"/>
    </schema>
</types>
...
<message name="NilPartRequest">
    <part name="x" element="xsd1:test_string_x"/>
    <part name="y" element="xsd1:test_short_y"/>
</message>
<message name="NilPartResponse">
    <part name="return" element="xsd1:test_int_return"/>
    <part name="y" element="xsd1:test_short_y"/>
    <part name="z" element="xsd1:test_float_z"/>
</message>
...
<portType name="BasePortType">
    <operation name="send_receive_nil_part">
        <input name="doclit_nil_part_request"
                                 message="tns:NilPartRequest"/>
        <output name="doclit_nil_part_response"
                                 message="tns:NilPartResponse"/>
    </operation>
</portType>
...
```

Example

Example 221 shows how to use nillable atomic types, `IT_Bus::StringNillable`, `IT_Bus::ShortNillable`, `IT_Bus::IntNillable`, and `IT_Bus::FloatNillable`, in a simple Example.

Example 221: *Using Nillable Atomic Types as Operation Parameters*

```
// C++
IT_Bus::StringNillable  x("String for sending");
IT_Bus::ShortNillable  y(321);
IT_Bus::IntNillable  var_return;
IT_Bus::FloatNillable z;

try {
```

Example 221: *Using Nillable Atomic Types as Operation Parameters*

```
    // bc is a client proxy for the BasePortType port type.
    bc.send_receive_nil_part(x, y, var_return, z);
}
catch (IT_Bus::FaultException &ex) {
    // ... deal with the exception (not shown)
}

if (! y.is_nil()) { cout << "y = " << y.get() << endl; }
if (! z.is_nil()) { cout << "z = " << z.get() << endl; }

if (! var_return.is_nil()) {
    cout << "var_return = " << var_return.get() << endl;
}
```

The value of a nillable atomic type, `T`, can be initialized using either a constructor, `T()`, or the `T.set()` member function.

Before attempting to read the value of a nillable atomic type using `T.get()`, you should check that the value is non-nil using the `T.is_nil()` member function.

Nillable User-Defined Types

This subsection describes how to define and use nillable user-defined types. In C++, every user-defined type, *UserTypeName*, has a nillable counterpart, *UserTypeName*`Nillable`.

You can modify or access the value of a user-defined nillable type, `T`, using the `T.set()` and `T.get()` member functions, respectively. For full details of the API for nillable types see "C++ API for nillable types" on page 379.

WSDL example

Example 222 shows the definition of an XML schema `all` complex type, named `SOAPStruct`. This is a complex type with ordinary (that is, non-nillable) member elements.

Example 222: *WSDL Example of an All Complex Type*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BaseService"
   targetNamespace="http://soapinterop.org/"
    ...
    xmlns:tns="http://soapinterop.org/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://soapinterop.org/xsd">
    <types>
        <schema targetNamespace="http://soapinterop.org/xsd"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
          <complexType name="SOAPStruct">
              <all>
                  <element name="varFloat" type="xsd:float"/>
                  <element name="varInt" type="xsd:int"/>
                  <element name="varString" type="xsd:string"/>
```

Example 222: *WSDL Example of an All Complex Type*

```
              </all>
          </complexType>
            ...
        </schema>
    </types>
    ...
```

C++ mapping

Example 223 shows how the SOAPStruct type maps to C++. In addition to the regular mapping, which produces the C++ SOAPStruct and SOAPStructPtr classes, the WSDL-to-C++ compiler also generates a nillable type, SOAPStructNillable, and an associated smart pointer type, SOAPStructNillablePtr.

Example 223: *C++ Mapping of the SOAPStruct All Complex Type*

```
// C++
namespace INTEROP
{
    class SOAPStruct : public IT_Bus::AllComplexType  { ... }
    typedef IT_AutoPtr<SOAPStruct> SOAPStructPtr;

    typedef IT_Bus::NillableValue<SOAPStruct>
                SOAPStructNillable;
    typedef IT_Bus::NillablePtr<SOAPStruct>
                SOAPStructNillablePtr;
};
```

The API for the SOAPStructNillable type is defined in “C++ API for nillable types” on page 379.

Example

The following example shows how to initialize an instance of SOAPStructNillable type, s_nillable. The nillable type is created in two steps: first of all, a SOAPStruct instance, s, is initialized; then the SOAPStruct instance is used to initialize a SOAPStructNillable instance.

```
// C++
// Initialize a SOAPStruct instance.
INTEROP::SOAPStruct s;
s.setvarFloat(3.14);
s.setvarInt(1234);
s.setvarString("Hello world!");

// Initialize a SOAPStructNillable instance.
INTEROP::SOAPStructNillable s_nillable;
s_nillable.set(s);
```

The next example shows how to access the contents of the `SOAPStructNillable` type. Note that before attempting to access the value of the `SOAPStructNillable` using `get()`, you should check that the value is not nil using `is_nil()`.

```
// C++
if (! s_nillable.is_nil()) {
    cout << "varFloat =  " << s_nillable.get().getvarFloat()
         << endl;
    cout << "varInt =    " << s_nillable.get().getvarInt()
         << endl;
    cout << "varString = " << s_nillable.get().getvarString()
         << endl;
}
```

Nested Atomic Type Nillable Elements

This subsection describes how to define and use complex types (except arrays) that have some nillable member elements. That is, the type as a whole is not nillable, although some of its elements are.

The WSDL-to-C++ compiler treats a type with nillable elements as a special case. If a member element, *ElementName*, is defined with `xsd:nillable` equal to `true`, the element's C++ modifiers and accessors are then primarily pointer based.

For example, given that a member element *ElementName* is of *AtomicType* type, the accessors and modifier would have the following signatures:

```
const AtomicType * getElementName() const;
AtomicType *       getElementName();
void                  setElementName(const AtomicType * val);
```

And an additional convenience function that allows you to set an element value using pass-by-reference:

```
void                  setElementName(const AtomicType & val);
```

Note: Arrays with nillable elements are treated differently—see "Nillable Elements of an Array" on page 391.

WSDL example

Example 224 defines a sequence complex type, `Nil_SOAPStruct`, which has some nillable elements, `varInt`, `varFloat`, and `varString`.

Example 224: *WSDL Example of a Sequence Type with Nillable Elements*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BaseService"
    targetNamespace="http://soapinterop.org/"
    ...
    xmlns:tns="http://soapinterop.org/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://soapinterop.org/xsd">
    <types>
```

Example 224: *WSDL Example of a Sequence Type with Nillable Elements*

```
        <schema targetNamespace="http://soapinterop.org/xsd"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
            ...
          <complexType name="Nil_SOAPStruct">
              <sequence>
                  <element name="varInt" nillable="true"
                           type="xsd:int"/>
                  <element name="varFloat" nillable="true"
                           type="xsd:float"/>
                  <element name="varString" nillable="true"
                           type="xsd:string"/>
              </sequence>
          </complexType>
        </schema>
    </types>
    ...
```

C++ mapping

Example 225 shows how the `Nil_SOAPStruct` sequence complex type is mapped to C++. Note how the accessors for the nillable member elements, `get`*ElementName*`()`, return a pointer instead of a value; and how the modifiers for the nillable member elements, `set`*ElementName*`()`, take either a pointer argument or a reference argument. For example, the `getvarInt()` function returns a pointer to an `IT_Bus::Int` rather an `IT_Bus::Int` value.

Example 225: *C++ Mapping of the Nil_SOAPStruct Sequence Type*

```
// C++
namespace INTEROP {
    class Nil_SOAPStruct : public IT_Bus::SequenceComplexType
    {
      public:
        Nil_SOAPStruct();
        Nil_SOAPStruct(const Nil_SOAPStruct& copy);
        virtual ~Nil_SOAPStruct();
        ...
       const IT_Bus::Int * getvarInt() const;
       IT_Bus::Int * getvarInt();
       void setvarInt(const IT_Bus::Int * val);
       void setvarInt(const IT_Bus::Int & val);

       const IT_Bus::Float * getvarFloat() const;
       IT_Bus::Float *       getvarFloat();
       void setvarFloat(const IT_Bus::Float * val);
       void setvarFloat(const IT_Bus::Float & val);

       const IT_Bus::String * getvarString() const;
       IT_Bus::String *       getvarString();
       void setvarString(const IT_Bus::String * val);
       void setvarString(const IT_Bus::String & val);

        virtual const IT_Bus::QName & get_type() const;
        ...
```

Example 225: *C++ Mapping of the Nil_SOAPStruct Sequence Type*

```
    };

    typedef IT_AutoPtr<Nil_SOAPStruct> Nil_SOAPStructPtr;

    typedef IT_Bus::NillableValue<Nil_SOAPStruct, &Nil_SOAPStructQName>
   Nil_SOAPStructNillable;

    typedef IT_Bus::NillablePtr<Nil_SOAPStruct, &Nil_SOAPStructQName>
   Nil_SOAPStructNillablePtr;
    ...
};
```

Example

The following example shows how to create and initialize a `Nil_SOAPStruct` instance. Notice, for example, how the `setvarInt(const IT_Bus::Int&)` convenience function allows you to pass the integer argument as a reference, `i`, instead of a pointer.

```
// C++
Nil_SOAPStruct nil_s;

IT_Bus::Float f = 3.14;
IT_Bus::Int   i = 1234;
IT_Bus::String s = "A non-nil string.";

nil_s.setvarInt(i);
nil_s.setvarFloat(f);
nil_s.setvarString(s);
```

The next example shows how to read the nillable elements of the `Nil_SOAPStruct` instance. Note how the elements are checked for nilness by comparing the result of calling `get`*ElementName*`()` with `0`.

```
// C++
if (nil_s.getvarInt() != 0) {
    cout << "varInt = " << *nil_s.getvarInt() << endl;
}

if (nil_s.getvarFloat() != 0) {
    cout << "varFloat = " << *nil_s.getvarFloat() << endl;
}

if (nil_s.getvarString() != 0) {
    cout << "varString = " << *nil_s.getvarString() << endl;
}
```

Nested User-Defined Nillable Elements

This subsection describes how to define and use complex types that have nillable member elements of user-defined type.

The WSDL-to-C++ compiler treats user-defined nillable elements as a special case. As with nillable elements of atomic type, if a member element of user-defined type, *ElementName*, is defined with `xsd:nillable` equal to `true`, the element's C++ modifiers and accessors are then primarily pointer based.

For example, given that a member element *ElementName* is of *UserType* type, the accessors and modifier would have the following signatures:

```
const UserType * getElementName() const;
UserType *       getElementName();
void               setElementName(const UserType * val);
void               setElementName(const UserType & val);
```

Note: Arrays with nillable elements are treated differently—see "Nillable Elements of an Array" on page 391.

WSDL example

Example 226 defines a sequence complex type, `Nil_NestedSOAPStruct`, which includes a nillable element of `SOAPStruct` type, `varSOAP`.

Example 226: *WSDL Example of a Nillable All Type inside a Sequence Type*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BaseService"
   targetNamespace="http://soapinterop.org/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    ...
    xmlns:tns="http://soapinterop.org/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://soapinterop.org/xsd">
    <types>
        <schema targetNamespace="http://soapinterop.org/xsd"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
            <complexType name="SOAPStruct">
                <all>
                    <element name="varFloat" type="xsd:float"/>
                    <element name="varInt" type="xsd:int"/>
                  <element name="varString" type="xsd:string"/>
                </all>
            </complexType>
            ...
          <complexType name="Nil_NestedSOAPStruct">
              <sequence>
                  <element name="varInt" nillable="true"
                          type="xsd:int"/>
                  <element name="varSOAP" nillable="true"
                          type="xsd1:SOAPStruct"/>
              </sequence>
          </complexType>
            ...
        </schema>
    </types>
    ...
```

C++ mapping

Example 227 shows how the `Nil_NestedSOAPStruct` sequence complex type is mapped to C++. Note how the `getvarSOAP()` functions return a pointer to a `SOAPStruct` rather than a `SOAPStruct` value.

Example 227: *C++ Mapping of the Nil_NestedSOAPStruct Type*

```
// C++
class Nil_NestedSOAPStruct : public IT_Bus::SequenceComplexType
{
  public:
    Nil_NestedSOAPStruct();
    Nil_NestedSOAPStruct(const Nil_NestedSOAPStruct& copy);
    virtual ~Nil_NestedSOAPStruct();
    ...
    const IT_Bus::Int * getvarInt() const;
    IT_Bus::Int *       getvarInt();
    void setvarInt(const IT_Bus::Int * val);
    void setvarInt(const IT_Bus::Int & val);

    const SOAPStruct * getvarSOAP() const;
    SOAPStruct *       getvarSOAP();
    void setvarSOAP(const SOAPStruct * val);
    void setvarSOAP(const SOAPStruct & val);

    virtual const IT_Bus::QName & get_type() const;
    ...
};
```

NillablePtr types

To help you manage the memory associated with nillable elements of user-defined type, *UserType*, the WSDL-to-C++ utility generates a nillable smart pointer type, *UserType*`NillablePtr`. The `NillablePtr` template types are similar to the `std::auto_ptr<>` template types from the Standard Template Library—see “Smart Pointers”.

For example, the following extract from the generated *WSDLFileName*`_wsdlTypes.h` header file defines a `SOAPStructNillablePtr` type, which is used to represent `SOAPStruct` nillable pointers:

```
// C++
typedef IT_Bus::NillablePtr<SOAPStruct, &SOAPStructQName>
   SOAPStructNillablePtr;
```

Example 228 shows the API for the `NillablePtr` template class. A `NillablePtr` instance can be initialized using either a `NillablePtr()` constructor, a `set()` member function, or an `operator=()` assignment operator. The `is_nil()` member function tests the pointer for nilness.

Example 228: *The NillablePtr Template Class*

```
// C++
namespace IT_Bus
{
    /**
     * Template implementation of Nillable as an auto_ptr.
     * T is the C++ type of data, TYPE is the data type qname.
     */
    template <class T, const QName* TYPE>
    class NillablePtr : public Nillable, public IT_AutoPtr<T>
    {
      public:
        NillablePtr();
        NillablePtr(const NillablePtr& other);
        NillablePtr(T* data);
        virtual ~NillablePtr();
        ...
        void set(const T* data);

        virtual Boolean is_nil() const;

        virtual const QName& get_type() const;
        ...
    };
    ...
};
```

Example

The following example shows how to create and initialize a `Nil_NestedSOAPStruct` instance. Notice how the argument to `setvarSOAP()` is passed as a pointer, `&nillable_struct`.

```
// C++
// Construct a smart nillable pointer.
// The SOAPStruct memory is owned by the smart nillable pointer.
SOAPStruct nillable_struct;
nillable_struct.setvarFloat(3.14);
nillable_struct.setvarInt(4321);
nillable_struct.setvarString("Nillable struct element.");

// Construct a nested struct.
Nil_NestedSOAPStruct outer_struct;
IT_Bus::Int k = 4321
outer_struct.setvarInt(&k);

// MEMORY MANAGEMENT: The argument to setvarSOAP is deep
   copied.
outer_struct.setvarSOAP(&nillable_struct);
```

The next example shows how to read the nillable elements of the `Nil_NestedSOAPStruct` instance. Note how the `varSOAP` element is checked for nilness by calling `is_nil()`.

```
// C++
IT_Bus::Int * int_p  = outer_struct.getvarInt();

// MEMORY MANAGEMENT: outer_struct owns the return value.
SOAPStruct *  nillable_struct_p = outer_struct.getvarSOAP();

if (int_p != 0) {
    cout << "varInt = " << *int_p << endl;
}

if (!nillable_struct_p.is_nil() ) {
    cout << "varSOAP = " << *nillable_struct_p << endl;
}
```

Nillable Elements of an Array

This subsection describes how to define and use array complex types with nillable array elements. To define an array with nillable elements, add a `nillable="true"` setting to the array element declaration.

An array with nillable elements has the following general syntax:

```
<complexType name="ArrayName">
    <sequence>
        <element name="ElemName" type="ElemType" nillable="true"
                 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*.

WSDL example

Example 229 shows defines an array complex type, `Nil_SOAPArray` (the name indicates that the type is used in a SOAP example, not that it is defined using SOAP array syntax) which has nillable array elements, `item`.

Example 229:*WSDL Example of an Array with Nillable Elements*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BaseService" targetNamespace="http://soapinterop.org/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://soapinterop.org/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://soapinterop.org/xsd">
    <types>
        <schema targetNamespace="http://soapinterop.org/xsd"
            xmlns="http://www.w3.org/2001/XMLSchema"
```

Example 229: *WSDL Example of an Array with Nillable Elements*

```
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
            ...
           <complexType name="Nil_SOAPArray">
               <sequence>
                   <element name="item" nillable="true"
                            type="xsd:short" minOccurs="10"
                            maxOccurs="10"/>
               </sequence>
           </complexType>
            ...
        </schema>
    </types>
    ...
```

C++ mapping

Example 230 shows how the `Nil_SOAPArray` array complex type is mapped to C++. Note that the array elements are of `IT_Bus::ShortNillable` type.

Example 230: *C++ Mapping of the Nil_SOAPArray Array Type*

```
// C++
namespace INTEROP {
    class Nil_SOAPArray
      : public IT_Bus::ArrayT<IT_Bus::ShortNillable, &Nil_SOAPArray_item_qname, 10, 10>
    {
      public:
        Nil_SOAPArray();
        Nil_SOAPArray(const Nil_SOAPArray& copy);
        Nil_SOAPArray(size_t dimensions[]);
        Nil_SOAPArray(size_t dimension0);
        virtual ~Nil_SOAPArray();

        ...
        const IT_Bus::ElementListT<IT_Bus::ShortNillable> &
        getitem() const;

        IT_Bus::ElementListT<IT_Bus::ShortNillable> &
        getitem();

        void
        setitem(const IT_Vector<IT_Bus::ShortNillable> & val);

        virtual const IT_Bus::QName &
        get_type() const;
    };

    typedef IT_AutoPtr<Nil_SOAPArray> Nil_SOAPArrayPtr;

   typedef IT_Bus::NillableValue<Nil_SOAPArray, &Nil_SOAPArrayQName> Nil_SOAPArrayNillable;

    typedef IT_Bus::NillablePtr<Nil_SOAPArray, &Nil_SOAPArrayQName>
   Nil_SOAPArrayNillablePtr;
};
```

Example

The following C++ example shows how to create and initialize a `Nil_SOAPArray` instance. Because each array element is of `IT_Bus::ShortNillable` type, the array elements must be initialized using the `set()` member function. Any elements not explicitly initialized are nil by default.

```
// C++
Nil_SOAPArray nil_s(10);
nil_s[0].set(10);
nil_s[1].set(20);
nil_s[2].set(30);
nil_s[3].set(40);
nil_s[4].set(50);
// The remaining five element values are left as nil.
```

The next C++ example shows how to access the nillable array elements. You should check each of the array elements for nilness using the `is_nil()` member function before attempting to read an array element value.

```
// C++
for (size_t i=0; i<10; i++) {
    if (! nil_s[i].is_nil()) {
        cout << "Nil_SOAPArray[" << i << "] = "
             << nil_s[i].get() << endl;
    }
}
```

Substitution Groups

The XML syntax for defining a *substitution group* enables you to define a relationship between XML elements, which is analogous to the inheritance relationship between XML data types.

For example, Figure 30 shows an inheritance tree of data types next to a parallel inheritance tree of elements. The type inheritance tree consists of a base type, `BuildingType`, and two derived (by extension) types, `HouseType` and `ApartmentBlockType`. The element inheritance tree consists of a *head element*, `building`, and two *substitute elements*, `house` and `apartmentBlock`.

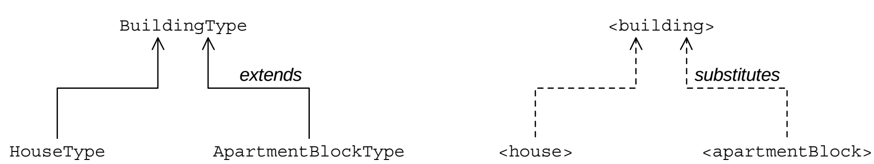

Figure 30: *Relationship Between Elements in a Substitution Group*

> **Note:** Substitution groups are currently supported only by the SOAP binding.

Defining a substitution group

You can define an XML substitution group as follows:

1. Define a *head element* (for example, `xsd1:building`) directly within a `<schema>` scope. The head element plays a role analogous to that of a base type in an inheritance tree—other elements can be defined to substitute the head element.
2. Define one or more *substitute elements* (for example, `xsd1:house` and `xsd1:apartmentBlock`) directly within a `<schema>` scope, setting the `substitutionGroup` attribute to the head element's QName—for example:

   ```
   <element name="house" type="xsd1:HouseType"
           substitutionGroup="xsd1:building" />
   ```

   A substitute element plays a role analogous to that of a sub-type in an inheritance tree—the substitute element can be used in place of the head element.

   Note: A substitute element must be of the same type as or be derived from the head element type.

3. Define a complex type (for example, a sequence group, all group, or choice group) that includes a reference to the head element. To define an element reference, use the `ref` attribute.

   For example, the following `PropertyType` type includes a reference to the `building` head element. In this case, the element with the `ref` attribute is called a *substitutable element*.

   ```
   <complexType name="PropertyType">
       <sequence>
           <element ref="xsd1:building"/>
           <element name="site" type="xsd1:SiteType"/>
       </sequence>
   </complexType>
   ```

   Note: Currently, Artix does *not* support substitutable elements in an `<all>` complex type.

XSD example

Example 231 shows the definition of a sequence group, `PropertyType`, that includes a single substitutable element, `xsd1:building`.

Example 231: *Sequence Type Containing a Substitutable Element*

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd1="http://schemas.iona.com/realestate"
        targetNamespace="http://schemas.iona.com/realestate">

    <!-- Type definitions -->

    <complexType name="BuildingType">
```

Example 231: *Sequence Type Containing a Substitutable Element*

```
        <sequence>
            <element name="squareMeters" type="xsd:int"/>
        </sequence>
    </complexType>

    <complexType name="HouseType">
        <complexContent>
            <extension base="xsd1:BuildingType">
                <sequence>
                  <element name="houseKind" type="xsd:string"/>
                </sequence>
            </extension>
        </complexContent>
    </complexType>

    <complexType name="ApartmentBlockType">
        <complexContent>
            <extension base="xsd1:BuildingType">
                <sequence>
                   <element name="nApartments" type="xsd:int"/>
                </sequence>
            </extension>
        </complexContent>
    </complexType>

    <!-- Global Elements -->

    <element name="building" type="xsd1:BuildingType"/>

    <element name="house"
        type="xsd1:HouseType"
        substitutionGroup="building"
        final="#all"/>

    <element name="apartmentBlock"
        type="xsd1:ApartmentBlockType"
       substitutionGroup="building"
        final="#all"/>

    <!-- More Types -->

    <complexType name="SiteType">
        <sequence>
            <element name="squareMeters" type="xsd:int"/>
        </sequence>
    </complexType>

    <complexType name="PropertyType">
        <sequence>
            <element ref="xsd1:building"/>
            <element name="site" type="xsd1:SiteType"/>
        </sequence>
    </complexType>

</schema>
```

The substitution group consists of the following elements:

- The head element, `xsd1:building`, and
- The substitute elements, `xsd1:house` and `xsd1:apartmentBlock`.

Substitutable element appearing in a sequence group

Example 232 shows how the `PropertyType` sequence group from Example 231 on page 394 maps to C++.

Example 232: *C++ Mapping of PropertyType Sequence Type*

```
// C++

namespace COM_IONA_SCHEMAS_REALESTATE
{
    class PropertyType
      : public IT_Bus::SequenceComplexType,
        public IT_Bus::ComplexTypeWithSubstitution
    {
      public:
        ...

       enum buildingDiscriminator
       {
           building_enum,
           house_enum,
           apartmentBlock_enum,
           building_MAXLONG=-1
       } var_buildingDiscriminator;

        buildingDiscriminator get_buildingDiscriminator() const
        {
            return var_buildingDiscriminator;
        }

        IT_Bus::UInt get_buildingDiscriminator_as_uint() const
        {
            return var_buildingDiscriminator;
        }

        BuildingType &       getbuilding();
        const BuildingType & getbuilding() const;
        void setbuilding(const BuildingType & val);

        HouseType &       gethouse();
        const HouseType & gethouse() const;
        void sethouse(const HouseType & val);

        ApartmentBlockType &       getapartmentBlock();
        const ApartmentBlockType & getapartmentBlock() const;
        void setapartmentBlock(const ApartmentBlockType & val);

        SiteType &       getsite();
        const SiteType & getsite() const;
        void setsite(const SiteType & val);
```

Example 232: *C++ Mapping of PropertyType Sequence Type*

```
      private:
        ...
    };
    ...
}
```

For each substitutable element appearing in a sequence group, the WSDL-to-C++ compiler generates the following enumeration type and discriminator functions:

```
// C++

enum HeadElementDiscriminator {
    ...
} var_HeadElementDiscriminator;

HeadElementDiscriminator get_HeadElementDiscriminator();

IT_Bus::UInt get_HeadElementDiscriminator();
```

Where *HeadElement* is the local part of the head element QName. The value returned by `get_HeadElementDiscriminator()` tells you what kind of element is currently stored as the substitutable element. You must check the discriminator value prior to calling `getElementName()` for an element belonging to the *HeadElement* substitution group.

Substitutable element appearing in a choice group

You can include a substitutable element in a choice group. The choice group mapping is, however, different from the sequence group mapping. Because a choice group already includes a discriminator when mapped to C++, the substitution group enumerations are simply absorbed into the existing choice enumeration.

For example, Example 233 redefines `PropertyChoiceType` as a *choice group* that contains a single substitutable element, `xsd1:building`.

Example 233: *Choice Type Containing a Substitutable Element*

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd1="http://schemas.iona.com/realestate"
        targetNamespace="http://schemas.iona.com/realestate">

    ...

    <complexType name="PropertyChoiceType">
        <choice>
            <element ref="xsd1:building"/>
            <element name="site" type="xsd1:SiteType"/>
        </choice>
    </complexType>
```

Example 233: *Choice Type Containing a Substitutable Element*

```
</schema>
```

The `PropertyChoiceType` choice group defined in the preceding Example 233 maps to the C++ `PropertyChoiceType` class shown in Example 234.

Example 234: *C++ Mapping of the PropertyChoiceType Choice Group*

```
// C++
namespace COM_IONA_SCHEMAS_REALESTATE
{
    class PropertyChoiceType : public IT_Bus::ChoiceComplexType
    {
      public:
        ...

       enum PropertyChoiceTypeDiscriminator
       {
           building_enum,
           house_enum,
           apartmentBlock_enum,
           site_enum,
           PropertyChoiceType_MAXLONG=-1
       } m_discriminator;

        PropertyChoiceTypeDiscriminator get_discriminator() const
        {
            return m_discriminator;
        }

        IT_Bus::UInt get_discriminator_as_uint() const
        {
            return m_discriminator;
        }

        // Get and Set functions (not shown)
        ...

      private:
        ...
    };
}
```

For the `PropertyChoiceType` choice group, the WSDL-to-C++ compiler generates a single enumeration type, `PropertyChoiceTypeDiscriminator`, and discriminator functions, `get_discriminator()` and `get_discriminator_as_uint()`.

In general, when mapping a choice group, the alternatives for all of the substitutable elements and all of the regular elements in the choice group are consolidated into a single enumeration type.

Substitutable element with occurrence constraints

You can add occurrence constraints to a substitutable element. For example, the `MultiPropertyType` defined in Example 235 contains an unbounded number of `building` elements.

Example 235: *Substitutable Element with Occurrence Constraints*

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd1="http://schemas.iona.com/realestate"
        targetNamespace="http://schemas.iona.com/realestate">

    ...

    <complexType name="MultiPropertyType">
        <sequence>
            <element ref="xsd1:building"
                     minOccurs="1" maxOccurs="unbounded" />
            <element name="site" type="xsd1:SiteType"/>
        </sequence>
    </complexType>

    <element name="MultiProperty"
             type="xsd1:MultiPropertyType"/>

</schema>
```

The array of substitutable elements appearing in `MultiPropertyType` need not be all of one type; they can be mixed. For example, the following would be a valid instance of `<MultiProperty>`:

```
<MultiProperty>
    <house> ... </house>
    <apartmentBlock> ... </apartmentBlock>
    <house> ... </house>
    <apartmentBlock> ... </apartmentBlock>

    <site> ... </site>
</MultiProperty>
```

The discriminator returned from `get_buildingDiscriminator()` is interpreted as follows:

- `MultiPropertyType::house_enum`

  An array consisting exclusively of `house` elements. Use the `gethouse()` function to obtain the element list, of `IT_Bus::ElementListT<HouseType>` type.
- `MultiPropertyType::apartmentBlock_enum`

  An array consists exclusively of `apartmentBlock` elements. Use the `getapartmentBlock()` function to obtain the element list, of `IT_Bus::ElementListT<ApartmentBlockType>` type.
- `MultiPropertyType::building_enum`

  A mixed array. Use the `getbuilding()` function to obtain the element list, of `IT_Bus::ElementListT<BuildingType>` type. To determine the actual type of each array element, attempt to downcast to one of the types in the substitution group (HouseType or ApartmentBlockType).

For more details about element lists, see "Element Occurrence Constraints" on page 366.

Abstract head element

You can define the head element to be *abstract*. An abstract head element is analogous to an abstract base class—that is, it cannot be used directly, but serves only as a basis for defining substitute elements. You can make a head element abstract by setting the abstract attribute to true in the element definition.

For example, the xsd1:building head element from Example 231 on page 394 can be declared abstract as follows:

```
<element name="building" type="xsd1:BuildingType"
         abstract="true"/>
```

When this modified version of the XML schema is compiled into C++, the generated PropertyType class omits the getbuilding() and setbuilding() functions. The PropertyType::building_enum value is also omitted from the buildingDiscriminator enumeration type. In other words, the only elements you can use for the substitutable element in the PropertyType are the house or apartmentBlock elements.

Note: An exception to this mapping rule occurs when a substitution element is defined with *occurrence constraints*. For example, if building is declared abstract, the MultiPropertyType would include the getbuilding() and setbuilding() functions when mapped to C++. These functions are needed to access and modify mixed arrays. It is still forbidden to include building elements directly in the array, however.

SOAP Arrays

In addition to the basic array types described in "Arrays" on page 334, Artix also provides support for SOAP arrays. SOAP arrays have a relatively rich feature set, including support for *sparse arrays* and *partially transmitted arrays*. Consequently, Artix implements a distinct C++ mapping specifically for SOAP arrays, which is different from the C++ mapping described in the "Arrays" section.

This section contains the following subsections:

- Introduction to SOAP Arrays
- Multi-Dimensional Arrays
- Sparse Arrays
- Partially Transmitted Arrays

Introduction to SOAP Arrays

This section describes the syntax for defining SOAP arrays in WSDL and discusses how to program a simple one-dimensional array of strings. The following topics are discussed:

- Syntax.
- C++ mapping.
- Definition of a one-dimensional SOAP array.
- Sample encoding.
- Example.

Syntax

In general, SOAP array types are defined by deriving from the `SOAP-ENC:Array` base type (deriving by restriction). The type definition must conform to the following syntax:

```
<complexType name="<SOAPArrayType>">
    <complexContent>
        <restriction base="SOAP-ENC:Array">
            <attribute ref="SOAP-ENC:arrayType"
             wsdl:arrayType="<ElementType><ArrayBounds>"/>
        </restriction>
    </complexContent>
</complexType>
```

Where *<SOAPArrayType>* is the name of the newly-defined array type, *<ElementType>* specifies the type of the array elements (for example, `xsd:int`, `xsd:string`, or a user type), and *<ArrayBounds>* specifies the dimensions of the array (for example, `[]`, `[,]`, `[,,]`, `[,][]`, `[,,][]`, `[,][][]`, and so on). The `SOAP-ENC` namespace prefix maps to the `http://schemas.xmlsoap.org/soap/encoding/` namespace URI and the `wsdl` namespace prefix maps to the `http://schemas.xmlsoap.org/wsdl/` namespace URI.

Note: In the current version of Artix, the preceding syntax is the *only* case where derivation from a complex type is supported. Definition of a SOAP array is treated as a special case.

C++ mapping

A given *SOAPArrayType* array maps to a C++ class of the same name, which inherits from the `IT_Bus::SoapEncArrayT<>` template class. The *SOAPArrayType* C++ class overloads the `[]` operator to provide access to the array elements. The size of the array is returned by the `get_extents()` member function.

Definition of a one-dimensional SOAP array

Example 236 shows how to define a one-dimensional array of strings, `ArrayOfSOAPString`, as a SOAP array. The `wsdl:arrayType` attribute specifies the type of the array elements, `xsd:string`, and the number of dimensions, `[]` implying one dimension.

Example 236:*Definition of the ArrayOfSOAPString SOAP Array*

```
<definitions name="BaseService" targetNamespace="http://soapinterop.org/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://soapinterop.org/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://soapinterop.org/xsd">
    <types>
        <schema targetNamespace="http://soapinterop.org/xsd"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
           <complexType name="ArrayOfSOAPString">
               <complexContent>
                   <restriction base="SOAP-ENC:Array">
                       <attribute ref="SOAP-ENC:arrayType"
                           wsdl:arrayType="xsd:string[]"/>
                   </restriction>
               </complexContent>
           </complexType>
            ...
</definitions>
```

Sample encoding

Example 237 shows the encoding of a sample `ArrayOfSOAPString` instance, which is how the array instance might look when transmitted as part of a WSDL operation.

Example 237:*Sample Encoding of ArrayOfSOAPString*

```
1 <ArrayOfSOAPString SOAP-ENC:arrayType="xsd:string[2]">
2     <item>Hello</item>
      <item>world!</item>
  </ArrayOfSOAPString>
```

The preceding WSDL fragment can be explained as follows:

1. The element type and the array size are specified by the `SOAP-ENC:arrayType` attribute. Because `ArrayOfSOAPString` has been derived by restriction, `SOAP-ENC:arrayType` can only have values of the form `xsd:string[`*ArraySize*`]`.
2. The XML elements that delimit the individual array values, for example `item`, can have an arbitrary name. These element names are not significant.

Example

Example 238 shows a C++ example of how to allocate and initialize an `ArrayOfSOAPString` instance with four elements.

Example 238: *Example of Initializing an ArrayOfSOAPString Instance*

```
// C++
// Allocate SOAP array of String
const size_t extents[] = {4};
1 ArrayOfSOAPString a_str(extents);

2 a_str[0] = "Hello";
a_str[1] = "to";
a_str[2] = "the";
a_str[3] = "world!";
```

The preceding C++ example can be explained as follows:

1. To specify the array's size, you pass a list of extents (of `size_t[]` type) to the `ArrayOfSOAPString` constructor. This style of constructor has the advantage that it is easily extended to the case of multi-dimensional arrays—see "Multi-Dimensional Arrays" on page 403.
2. The overloaded `[]` operator provides read/write access to individual array elements.

> **Note:** Be sure to initialize *every* element in the array, unless you want to create a sparse array (see "Sparse Arrays" on page 405). There are no default element values. Uninitialized elements are flagged as empty.

Multi-Dimensional Arrays

The syntax for SOAP arrays allows you to define the dimensions of a multi-dimensional array using two slightly different syntaxes:

- A comma-separated list between square brackets, for example `[,]` and `[,,]`.
- Multiple square brackets, for example `[][]` and `[][][]`.

Artix makes no distinction between the two styles of array definition. In both cases, the array is flattened for transmission and the C++ mapping is the same.

Definition of multi-dimensional SOAP array

Example 239 shows how to define a two-dimensional array of integers, `Array2OfInt`, as a SOAP array. The `wsdl:arrayType` attribute specifies the type of the array elements, `xsd:int`, and the number of dimensions, `[,]` implying an array of two dimensions.

Example 239: *Definition of the Array2OfInt SOAP Array*

```
<definitions ... >
    <types>
```

Example 239: *Definition of the Array2OfInt SOAP Array*

```
        <schema ... >
          <complexType name="Array2OfInt">
            <complexContent>
              <restriction base="SOAP-ENC:Array">
                <attribute ref="SOAP-ENC:arrayType"
                  wsdl:arrayType="xsd:int[,]"/>
              </restriction>
            </complexContent>
          </complexType>
        ...
</definitions>
```

Sample encoding of multi-dimensional SOAP array

Example 240 shows the encoding of a sample Array2OfInt instance, which is how the array instance might look when transmitted as part of a WSDL operation.

Example 240: *Sample Encoding of an Array2OfInt SOAP Array*

```
<Array2OfInt SOAP-ENC:arrayType="xsd:int[2,3]">
    <i>1</i>
    <i>2</i>
    <i>3</i>
    <i>4</i>
    <i>5</i>
    <i>6</i>
</Array2OfInt>
```

The dimensions of this array instance are specified as [2,3], giving a total of six elements. Notice that the encoded array is effectively flat, because no distinction is made between rows and columns of the two-dimensional array.

Given an array instance with dimensions, [I_MAX,J_MAX], a particular position in the array, [i,j], corresponds with the i*J_MAX+j element of the flattened array. In other words, the right most index of [i,j,...,k] is the fastest changing as you iterate over the elements of a flattened array.

Example of a multi-dimensional SOAP array

Example 241 shows a C++ example of how to allocate and initialize an Array2OfInt instance with dimensions, [2,3].

Example 241: *Initializing an Array2OfInt SOAP Array*

```
// C++
1 const size_t extents2[] = {2, 3};
Array2OfInt a2_soap(extents2);

size_t position[2];
```

Example 241:*Initializing an Array2OfInt SOAP Array*

```
2 size_t i_max = a2_soap.get_extents()[0];
  size_t j_max = a2_soap.get_extents()[1];
  for (size_t i=0; i<i_max; i++) {
      position[0] = i;
      for (size_t j=0; j<j_max; j++) {
          position[1] = j;
3         a2_soap[position] = (IT_Bus::Int) (i+1)*(j+1);
      }
  }
```

The preceding C++ example can be explained as follows:

1. The dimensions of this array instance are specified to be `[2,3]` by initializing an array of extents, of `size_t[]` type, and passing this array to the `Array2OfInt` constructor.
2. The dimensions of the `a2_soap` array can be retrieved by calling the `get_extents()` function, which returns an extents array that converts to `size_t[]` type.
3. The operator `[]` is overloaded on `Array2OfInt` to accept an argument of `size_t[]` type, which contains a list of indices specifying a particular array element.

Sparse Arrays

Sparse arrays are fully supported in Artix. Every SOAP array instance stores an array of status flags, one flag for each array element. The status of each array element is initially empty, flipping to non-empty the first time an array element is accessed or initialized.

Note: Sparse arrays are *not* optimized for minimization of storage space. Hence, a sparse array with dimensions `[1000,1000]` would always allocate storage for one million elements, irrespective of how many elements in the array are actually non-empty.

WARNING: Sparse arrays have been deprecated in the SOAP 1.2 specification. Hence, it is better to avoid using sparse arrays if possible.

Sample encoding

Example 242 shows the encoding of a sparse `Array2OfInt` instance, which is how the array instance might look when transmitted as part of a WSDL operation.

Example 242:*Sample Encoding of a Sparse Array2OfInt SOAP Array*

```
<Array2OfInt SOAP-ENC:arrayType="xsd:int[10,10]">
    <item SOAP-ENC:position="[3,0]">30</item>
    <item SOAP-ENC:position="[2,1]">21</item>
    <item SOAP-ENC:position="[1,2]">12</item>
    <item SOAP-ENC:position="[0,3]">3</item>
</Array2OfInt>
```

The array instance is defined to have the dimensions [10,10]. Out of a maximum 100 elements, only four, that is [3,0], [2,1], [1,2], and [0,3], are transmitted. When transmitting an array as a sparse array, the SOAP-ENC:position attribute enables you to specify the indices of each transmitted array element.

Initializing a sparse array

Example 243 shows an example of how to initialize a sparse array of Array2OfInt type.

Example 243:*Initializing a Sparse Array2OfInt SOAP Array*

```
// C++
const size_t extents2[] = {10, 10};
Array2OfInt a2_soap(extents2);

size_t position[2];

position[0] = 3;
position[1] = 0;
a2_soap[position] = 30;

position[0] = 2;
position[1] = 1;
a2_soap[position] = 21;

position[0] = 1;
position[1] = 2;
a2_soap[position] = 12;

position[0] = 0;
position[1] = 3;
a2_soap[position] = 3;
```

This example does not differ much from the case of initializing an ordinary non-sparse array (compare, for example, Example 241 on page 404). The only significant difference is that the majority of array elements are not initialized, hence they are flagged as empty by default.

Note: The state of an array element flips from empty to *non-empty* the first time it is accessed using the [] operator. Hence, attempting to read the value of an uninitialized array element can have the unintended side effect of flipping the array element status.

Reading a sparse array

Example 244 shows an example of how to read a sparse array of `Array2OfInt` type.

Example 244: *Reading a Sparse Array2OfInt SOAP Array*

```
// C++
...
size_t p2[2];
1 size_t i_max = a2_out.get_extents()[0];
size_t j_max = a2_out.get_extents()[1];
for (size_t i=0; i<i_max; i++) {
    p2[0] = i;
    for (size_t j=0; j<j_max; j++) {
        p2[1] = j;
2       if (!a2_out.is_empty(p2)) {
            cout << "a[" << i << "][" << j << "] = "
                 << a2_out[p2] << endl;
        }
    }
}
```

The preceding C++ example can be explained as follows:

1. The `get_extents()` function returns the full dimensions of the array (as a `size_t[]` array), irrespective of the actual number of non-empty elements in the sparse array.
2. Before attempting to read the value of an element in the sparse array, you should call the `is_empty()` function to check whether the particular array element exists or not.

   If you were to access all the elements of the array, irrespective of their status, the empty array elements would all flip to the non-empty state. Hence, you would lose the information about which elements were transmitted in the sparse array.

Partially Transmitted Arrays

A partially transmitted array is essentially a special case of a sparse array, where the transmitted array elements form one or more contiguous blocks within the array. The start index and end index of each block can have any value.

The difference between a partially transmitted array and a sparse array is significant only at the level of encoding. From the Artix programmer's perspective, there is no significant distinction between partially transmitted arrays and sparse arrays.

Sample encoding

Example 245 shows the encoding of a partially transmitted `ArrayOfSOAPString` instance.

Example 245: *Sample Encoding of a Partially Transmitted ArrayOfSOAPString Array*

```
<ArrayOfSOAPString SOAP-ENC:arrayType="xsd:string[10]"
                   SOAP-ENC:offset="[2]">
   <item>The third element</item>
   <item>The fourth element</item>
   <item SOAP-ENC:position="[6]">The seventh element</item>
   <item>The eighth element</item>
</ArrayOfSOAPString>
```

In this example, only the third, fourth, seventh, and eighth elements of a ten-element string array are actually transmitted. The `SOAP-ENC:offset` attribute is used to specify the index of the first transmitted array element. The default value of `SOAP-ENC:offset` is `[0]`. The `SOAP-ENC:position` attribute specifies the start of a new block within the array. If an `item` element does not have a position attribute, it is assumed to represent the next element in the array.

IT_Vector Template Class

The `IT_Vector` template class is an implementation of `std::vector`. Hence, the functionality provided by `IT_Vector` should be familiar from the C++ Standard Template Library.

Introduction to IT_Vector

This section provides a brief introduction to programming with the `IT_Vector` template type, which is modelled on the `std::vector` template type from the C++ Standard Template Library (STL).

Differences between IT_Vector and std::vector

Although `IT_Vector` is modelled closely on the STL vector type, `std::vector`, there are some differences. In particular, `IT_Vector` does not provide the following types:

`IT_Vector<`*`T`*`>::allocator_type`

Where *T* is the vector's element type. Hence, the `IT_Vector` type does not support an `allocator_type` optional final argument in its constructors.

The `IT_Vector` type does *not* support the following operations:

```
!=, <
```

The member functions listed in Table 45 are *not* defined in IT_Vector.

Table 45: *Member Functions Not Defined in IT_Vector*

Function	Type of Operation
at()	Element access (with range check)
clear()	List operation
assign()	Assignment
resize()	Size and capacity
max_size()	

Although clear() is not defined, you can easily get the same effect for a vector, v, by calling erase() as follows:

```
v.erase(v.begin(), v.end());
```

This has the effect of erasing all the elements in v, leaving an array of size 0.

Basic usage of IT_Vector

The size() member function and the indexing operator [] is all that you need to perform basic manipulation of vectors. Example 246 shows how to use these basic vector operations to initialize an integer vector with the first one hundred integer squares.

Example 246:*Using Basic IT_Vector Operations to Initialize a Vector*

```
// C++
// Allocate a vector with 100 elements
IT_Vector<IT_Bus::Int> v(100);

for (size_t k=0; k < v.size(); k++) {
    v[k] = (IT_Bus::Int) k*k;
}
```

Iterators

Instead of indexing vector elements using the operator [], you can use a vector iterator. A vector iterator, of IT_Vector<*T*>::iterator type, gives you pointer-style access to a vector's elements. The following operations are supported by IT_Vector<*T*>::iterator:

```
++, --, *, =, ==, !=
```

An iterator instance remembers its current position within the element list. The iterator can advance to the next element using ++, step back to the previous element using --, and access the current element using *.

The IT_Vector template also provides a reverse iterator, of IT_Vector<*T*>::reverse_iterator type. The reverse iterator differs from the regular iterator in that it starts at the end of the element list and traverses the list backwards. That is the meanings of ++ and -- are reversed.

Example using iterators

Example 246 on page 409 can be written in a more idiomatic style using vector iterators, as shown in Example 247.

Example 247:*Using Iterators to Initialize a Vector*

```
// C++
// Allocate a vector with 100 elements
IT_Vector<IT_Bus::Int> v(100);

IT_Vector<IT_Bus::Int>::iterator p = v.begin();
IT_Bus k_int = 0;

while (p != v.end())
{
    *p = k_int*k_int;
    ++p;
    ++k_int;
}
```

Summary of IT_Vector Operations

This section provides a brief summary of the types and operations supported by the IT_Vector template type. Note that the set of supported types and operations differs slightly from std::vector. They are described in the following categories:

- Member types.
- Iterators.
- Element access.
- Stack operations.
- List operations.
- Other operations.

Member types

Table 46 lists the member types defined in IT_Vector<*T*>.

Table 46: *Member Types Defined in IT_Vector<T>*

Member Type	Description
value_type	Type of element.
size_type	Type of subscripts.

Table 46: *Member Types Defined in IT_Vector<T>*

Member Type	Description
`difference_type`	Type of difference between iterators.
`iterator`	Behaves like `value_type*`.
`const_iterator`	Behaves like `const value_type*`.
`reverse_iterator`	Iterates in reverse, like `value_type*`.
`const_reverse_iterator`	Iterates in reverse, like `const value_type*`.
`reference`	Behaves like `value_type&`.
`const_reference`	Behaves like `const value_type&`.

Iterators

Table 47 lists the `IT_Vector` member functions returning iterators.

Table 47: *Iterator Member Functions of IT_Vector<T>*

Iterator Member Function	Description
`begin()`	Points to first element.
`end()`	Points to last element.
`rbegin()`	Points to first element of reverse sequence.
`rend()`	Points to last element of reverse sequence.

Element access

Table 48 lists the `IT_Vector` element access operations.

Table 48: *Element Access Operations for IT_Vector<T>*

Element Access Operation	Description
`[]`	Subscripting, unchecked access.
`front()`	First element.
`back()`	Last element.

Stack operations

Table 49 lists the `IT_Vector` stack operations.

Table 49: *Stack Operations for IT_Vector<T>*

Stack Operation	Description
`push_back()`	Add to end.
`pop_back()`	Remove last element.

List operations

Table 50 lists the `IT_Vector` list operations.

Table 50: *List Operations for IT_Vector<T>*

List Operations	Description
`insert(p,x)`	Add `x` before `p`.
`insert(p,n,x)`	Add `n` copies of `x` before `p`.
`insert(first,last)`	Add elements from `[first:last[` before `p`.
`erase(p)`	Remove element at `p`.
`erase(first,last)`	Erase `[first:last[`.

Other operations

Table 51 lists the other operations supported by `IT_Vector`.

Table 51: *Other Operations for IT_Vector<T>*

Operation	Description
`size()`	Number of elements.
`empty()`	Is the container empty?
`capacity()`	Space allocated.
`reserve()`	Reserve space for future expansion.
`swap()`	Swap all the elements between two vectors.
`==`	Test vectors for equality (member-wise).

IT_HashMap Template Class

The `IT_HashMap` template class is an implementation of `std::map`. Hence, the functionality provided by `IT_HashMap` should be familiar from the C++ Standard Template Library.

Introduction to IT_HashMap

his section provides a brief introduction to programming with the IT_HashMap template type, which is modelled on the std::map template type from the C++ Standard Template Library (STL).

Differences between IT_HashMap and std::map

Although IT_HashMap is modelled closely on the STL map type, std::map, there are some differences.

The member functions listed in Table 52 are *not* defined in IT_HashMap.

Table 52: *Member Functions Not Defined in IT_Vector*

Function	Type of Operation
clear()	List operation.
value_comp() key_comp()	Comparison operations.
count() upper_bound() lower_bound() equal_range()	Map operations
max_size()	Size and capacity.

Although clear() is not defined, you can easily get the same effect for a map, v, by calling erase() as follows:

```
m.erase(m.begin(), m.end());
```

This has the effect of erasing all the elements in m, leaving a map of size 0.

Summary of IT_HashMap Operations

This section provides a brief summary of the types and operations supported by the IT_HashMap template type. Note that the set of supported types and operations differs slightly from std::map. They are described in the following categories:

- Member types.
- Iterators.
- Element access.
- Map operations.
- List operations.
- Other operations.

Member types

Table 53 lists the member types defined in `IT_HashMap<T>`.

Table 53: *Member Types Defined in IT_HashMap<T>*

Member Type	Description
key_type	Type of the hash key.
data_type	Type of the hash value.
value_type	Type of element—a (key, value) pair).
size_type	Type of subscripts.
difference_type	Type of difference between iterators.
iterator	Behaves like value_type*.
const_iterator	Behaves like const value_type*.
reference_type	Behaves like value_type&.
const_reference_type	Behaves like const value_type&.

Iterators

Table 54 lists the `IT_HashMap` member functions returning iterators.

Table 54: *Iterator Member Functions of IT_HashMap<T>*

Iterator Member Function	Description
begin()	Points to first element.
end()	Points to last element.

Element access

Table 55 lists the `IT_HashMap` element access operations.

Table 55: *Element Access Operations for IT_HashMap<T>*

Element Access Operation	Description
[]	Subscripting. Use a hash key as the subscript.

Map operations

Table 56 lists the `IT_HashMap` map operations.

Table 56: *Map Operations for IT_HashMap<T>*

Map Operation	Description
`find(k)`	Returns an iterator to the element with the key, `k`.

List operations

Table 57 lists the `IT_HashMap` list operations.

Table 57: *List Operations for IT_HashMap<T>*

List Operations	Description
`insert(v)`	Insert a (key, value) pair into the hash map.
`insert(first,last)`	Insert (key, value) pairs from `[first:last[` from the given sequence.
`erase(p)`	Remove element at `p`.
`erase(k)`	Remove element identified by the key, `k`.
`erase(first,last)`	Erase `[first:last[`.

Other operations

Table 58 lists the other operations supported by `IT_HashMap`.

Table 58: *Other Operations for IT_HashMap<T>*

Operation	Description
`size()`	Number of elements.
`empty()`	Is the container empty?
`swap()`	Swap all the elements between two hash maps.
`==`	Test hash maps for equality (member-wise).

Unsupported XML Schema Constructs in Artix

The following XML schema constructs are currently not supported in Artix:

- Built-in types:
  - `xs:NOTATION`
  - `xs:IDREF`
  - `xs:IDREFS`
  - `xs:ENTITY`

    - xs:ENTITIES
- id attribute on schema constructs, wherever it is applicable.
- xs:anyAttribute
    - Supported only for SOAP binding.
    - Not supported in xs:attributeGroup.
- xs:anySimpleType
- xs:attribute
    - form attribute.
- xs:complexType
    - mixed, final, and block attributes.
    - simpleContent/restriction.
    - complexContent/restriction.
- xs:element
    - final, block, fixed, default and abstract attributes.
- xs:field
- xs:group
    - minOccurs, maxOccurs on local groups.
    - all inside a group.
- xs:key
- xs:keyref
- xs:notation
- xs:redefine
- xs:selector
- xs:simpleType
    - Some facet restrictions.
    - final attribute.
- xs:unique

Artix IDL to C++ Mapping

This chapter describes how Artix maps IDL to C++; that is, the mapping that arises by converting IDL to WSDL (using the IDL-to-WSDL compiler) and then WSDL to C++ (using the WSDL-to-C++ compiler).

Introduction to IDL Mapping

This chapter gives an overview of the Artix IDL-to-C++ mapping. Mapping IDL to C++ in Artix is performed as a two step process, as follows:

1. Map the IDL to WSDL using the Artix IDL compiler. For example, you could map a file, `SampleIDL.idl`, to a WSDL contract, `SampleIDL.wsdl`, using the following command:

   ```
   idl -wsdl SampleIDL.idl
   ```

2. Map the generated WSDL contract to C++ using the WSDL-to-C++ compiler. For example, you could generate C++ stub code from the `SampleIDL.wsdl` file using the following command:

   ```
   wsdltocpp SampleIDL.wsdl
   ```

Alternative C++ mappings

If you are already familiar with CORBA technology, you will know that there is an existing standard for mapping IDL to C++ directly, which is defined by the Object Management Group (OMG). Hence, two alternatives exist for mapping IDL to C++, as follows:

- Artix IDL-to-C++ mapping—this is a two stage mapping, consisting of IDL-to-WSDL and WSDL-to-C++. It is a Micro Focus-proprietary mapping.
- CORBA IDL-to-C++ mapping—as specified in the OMG C++ Language Mapping document (http://www.omg.org). This mapping is used, for example, by Orbix.

These alternative approaches are illustrated in Figure 31.

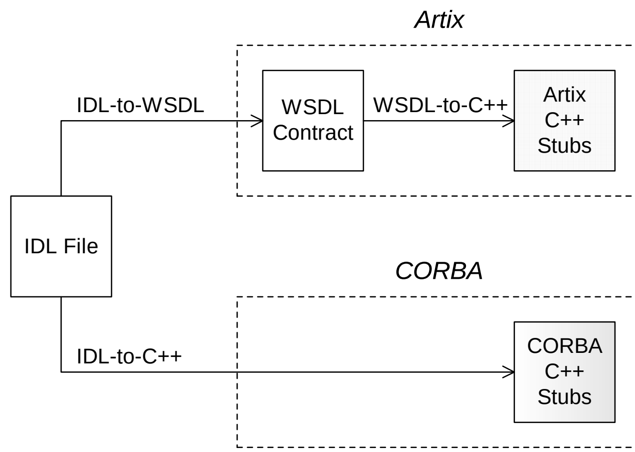

Figure 31: *Artix and CORBA Alternatives for IDL to C++ Mapping*

The advantage of using the Artix IDL-to-C++ mapping in an application is that it removes the CORBA dependency from your source code. For example, a server that implements an IDL interface using the Artix IDL-to-C++ mapping can interoperate with other Web service protocols, such as SOAP over HTTP.

Unsupported IDL types

The following IDL types are not supported by the Artix C++ mapping:

- `wchar`.
- `wstring`.
- `long double`.
- Value types.
- Boxed values.
- Local interfaces.
- Abstract interfaces.
- forward-declared interfaces.

IDL Basic Type Mapping

Table 59 shows how IDL basic types are mapped to WSDL and then to C++.

Table 59: *Artix Mapping of IDL Basic Types to C++*

IDL Type	WSDL Schema Type	C++ Type
`any`	`xsd:anyType`	`IT_Bus::AnyHolder`
`boolean`	`xsd:boolean`	`IT_Bus::Boolean`
`char`	`xsd:byte`	`IT_Bus::Byte`

Table 59: *Artix Mapping of IDL Basic Types to C++*

IDL Type	WSDL Schema Type	C++ Type
string	xsd:string	IT_Bus::String
wchar	xsd:string	IT_Bus::String
wstring	xsd:string	IT_Bus::String
short	xsd:short	IT_Bus::Short
long	xsd:int	IT_Bus::Int
long long	xsd:long	IT_Bus::Long
unsigned short	xsd:unsignedShort	IT_Bus::UShort
unsigned long	xsd:unsignedInt	IT_Bus::UInt
unsigned long long	xsd:unsignedLong	IT_Bus::ULong
float	xsd:float	IT_Bus::Float
double	xsd:double	IT_Bus::Double
long double	*Not supported*	*Not supported*
octet	xsd:unsignedByte	IT_Bus::UByte
fixed	xsd:decimal	IT_Bus::Decimal
Object	wsa:EndpointReferenceType	WS_Addressing::EndpointReferenceType

Mapping for string

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).

IDL Complex Type Mapping

This section describes how the following IDL data types are mapped to WSDL and then to C++:

- enum type.
- struct type.
- union type.
- sequence types.
- array types.
- exception types.
- typedef of a simple type.
- typedef of a complex type.

enum type

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

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

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>
```

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

```
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.

Programming with the Enumeration Type

For details of how to use the enumeration type in C++, see "Deriving Simple Types by Restriction" on page 299.

union type

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

```
// IDL
module SampleTypes {
    union Poly switch(short) {
        case 1: short theShort;
        case 2: string theString;
    };
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::Poly` union to an XML schema choice complex type, `SampleTypes.Poly`, as follows:

```
<xsd:complexType name="SampleTypes.Poly">
    <xsd:choice>
        <xsd:element name="theShort" type="xsd:short"/>
        <xsd:element name="theString" type="xsd:string"/>
    </xsd:choice>
</xsd:complexType>
```

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:
    ...
    const IT_Bus::Short gettheShort() const;
    void settheShort(const IT_Bus::Short& val);

    const IT_Bus::String& gettheString() const;
    void settheString(const IT_Bus::String& val);

    enum PolyDiscriminator
    {
        theShort,
        theString,
        Poly_MAXLONG=-1L
    } 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.

Programming with the Union Type

For details of how to use the union type in C++, see “Choice Complex Types” on page 311.

struct type

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

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

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>
```

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

```
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.

Programming with the Struct Type

For details of how to use the struct type in C++, see "Sequence Complex Types" on page 309.

sequence types

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

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

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>
```

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.

Programming with Sequence Types

For details of how to use sequence types in C++, see "Arrays" on page 334 and "IT_Vector Template Class" on page 408.

> **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.

array types

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

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

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>
```

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.

Programming with Array Types

For details of how to use array types in C++, see "Arrays" on page 334 and "IT_Vector Template Class" on page 408.

exception types

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

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

The IDL-to-WSDL compiler maps the SampleTypes::GenericExc exception to a WSDL sequence type, SampleTypes.GenericExc, and to a WSDL fault message, _exception.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="_exception.SampleTypes.GenericExc">
    <part name="exception" element="xsd1:SampleTypes.GenericExc"/>
</message>
```

The WSDL-to-C++ compiler maps the SampleTypes.GenericExc and _exception.SampleTypes.GenericExc types to C++ classes, SampleTypes_GenericExc and _exception_SampleTypes_GenericExc, 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 _exception_SampleTypes_GenericExcException : public IT_Bus::UserFaultException
{
  public:
    _exception_SampleTypes_GenericExcException();
    ...
    const SampleTypes_GenericExc & getexception() const;
    SampleTypes_GenericExc & getexception();
    void setexception(const SampleTypes_GenericExc & val);
    ...
};
```

Programming with Exceptions in Artix

For an example of how to initialize, throw and catch a WSDL fault exception, see "User-Defined Exceptions" on page 92.

typedef of a simple type

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

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

The IDL-to-WSDL compiler maps the SampleTypes::FloatAlias typedef directory to the type, xsd:float.

The WSDL-to-C++ compiler then maps the xsd:float type directly to the IT_Bus::Float C++ type. Hence, no C++ typedef is generated for the float type.

typedef of a complex type

Consider the following IDL typedef that defines an alias of a `struct`, `SampleTypes::SampleStructAlias`:

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

The IDL-to-WSDL compiler maps the `SampleTypes::SampleStructAlias` typedef directly to the plain, unaliased `SampleTypes.SampleStruct` type.

The WSDL-to-C++ compiler then maps the `SampleTypes.SampleStruct` WSDL type directly to the `SampleTypes::SampleStruct` C++ type. Hence, no C++ typedef is generated for this struct type. Instead of a typedef, the C++ mapping uses the original, unaliased type.

Note: The typedef of 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 Module and Interface Mapping

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 (see “Generating code from the command line” on page 484). 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 very 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.

See “Endpoint References” on page 119 for more details.

Operation mapping

Example 248 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 248:*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 248 on page 428, map to C++ as shown in Example 249,

Example 249:*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 250 shows two IDL attributes defined within the `SampleTypes::Foo` interface. The first attribute is readable and writable, `str_attr`, and the second attribute is readonly, `struct_attr`.

Example 250: *Example IDL Attributes*

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

The attributes from the preceding IDL, Example 250 on page 429, map to C++ as shown in Example 251,

Example 251: *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 readonly attribute, *AttributeName*, maps to a single accessor function in C++, `_get_`*AttributeName*`()`.

Reflection

Artix provides a reflection API which, analogously to Java reflection, enables you to unravel the structure of an Artix data type without having advance knowledge of it.

Introduction to Reflection

Artix reflection provides you with a way of representing Artix data types such that they are self-describing. Using the reflection API, you can employ recursive descent parsing to process any data type (whether built-in or user-defined), without knowing about the data type in advance.

The Artix reflection API is useful in those cases where you need to write general-purpose code to process Artix data types. If you are familiar with Java or CORBA, you probably recognize that Artix reflection offers functionality similar to that of Java reflection and CORBA DynamicAny.

C++ reflection class

In C++, reflection objects are represented by the `IT_Reflect::Reflection` base class and all of the classes derived from it—see “Overview of the Reflection API” on page 437 for more details.

Enabling reflection on generated classes

To enable reflection support on the C++ classes generated from XML schema types, you must pass the `-reflect` flag to the `wsdltocpp` utility.

Converting a user-defined type to a Reflection

To convert any XML schema type to an `IT_Bus::Reflection` instance, call one of the following `IT_Bus::AnyType::get_reflection()` functions:

```
// C++
IT_Reflect::Reflection*       get_reflection()
            IT_THROW_DECL((IT_Reflect::ReflectException));

const IT_Reflect::Reflection* get_reflection() const
            IT_THROW_DECL((IT_Reflect::ReflectException));
```

User-defined types always inherit from `IT_Bus::AnyType` and therefore also support the `get_reflection()` function.

Converting a built-in type to a Reflection

To convert a built-in type (such as `IT_Bus::Int`) to an `IT_Bus::Reflection` instance, construct an `IT_Reflect::ValueRef<T>` object (which inherits from `IT_Bus::Reflection`). For example, you can convert an integer, `IT_Bus::Int`, to a reflection object as follows:

```
// C++
IT_Bus::Int i = ...;
IT_Reflect::ValueRef<IT_Bus::Int> reflect_i(&i);
```

Converting a Reflection to an AnyType

To convert an `IT_Bus::Reflection` instance to an XML schema type (represented by the `IT_Bus::AnyType` base type), call one of the following `IT_Reflect::Reflection::get_reflected()` functions:

```
// C++
const IT_Bus::AnyType& get_reflected() const
   IT_THROW_DECL((ReflectException));

IT_Bus::AnyType&       get_reflected()
   IT_THROW_DECL((ReflectException));
```

Type descriptions

Currently, the Artix reflection API does *not* provide any data type that completely encapsulates an XML type description. However, some type information is implied in the structure of a `Reflection` object. In particular, `Reflection` objects support the `get_type_kind()` function, which has the following signature:

```
// C++
IT_Bus::AnyType::Kind get_type_kind() const
   IT_THROW_DECL((ReflectException));
```

The `IT_Bus::AnyType::Kind` type is an enumeration, defined as follows:

Example 252:*Definition of the IT_Bus::AnyType::Kind Enumeration*

```
// C++
namespace IT_Bus {
    class AnyType {
      public:
        enum Kind
        {
          NONE,  // AnyType::get_kind() will never return this.
          BUILT_IN,           // built-in type
          SIMPLE,             // simpleType restriction
          SEQUENCE,
          ALL,
          CHOICE,
          SIMPLE_CONTENT,
          ELEMENT_LIST,
          SOAP_ENC_ARRAY,
```

Example 252: *Definition of the IT_Bus::AnyType::Kind Enumeration*

```
            COMPLEX_CONTENT,
            NILLABLE,
            ANY_HOLDER,
            ANY,                // anyType restriction.
            ANY_LIST,
            SIMPLE_TYPE_LIST,
            SIMPLE_TYPE_UNION,
            TYPE_LIST,
          };
          ...
    };
};
```

Parsing reflection objects

The Artix reflection API is designed to let you parse the C++ representation of XML data types. Starting with an instance of a user-defined type in C++, you can convert this instance into an `IT_Bus::Reflection` instance (by calling `get_reflection()`) and use recursive descent parsing to process the returned reflection instance.

For example, you could use this functionality to print out the contents of an arbitrary Artix data type (see "Reflection Example" on page 455) or to convert an Artix data type into another data format.

The IT_Bus::Var Template Type

The `IT_Bus::Var<T>` template class is a smart pointer type that can be used to manage memory for reflection objects. Because functions in the reflection API generally return *pointers* to objects (which the caller is responsible for deleting), you have to exercise some care in order to avoid memory leaks.

The simplest way to manage memory for a reflection type, *T*, is to use the `IT_Bus::Var<T>` smart pointer type to reference the objects of type *T*. The `IT_Bus::Var<T>` type uses reference counting to manage the memory.

Reference counted objects

Objects referenced by `IT_Bus::Var<T>` must be *reference counted*. A reference counted object is an instance of a class that derives from `IT_Bus::RefCountedBase`, having the following properties:

- The initial reference count is 1.
- The reference count is incremented by calling `_add_ref()`.
- The reference count is decremented by calling `_remove_ref()`.
- When the reference count reaches zero, the object is deleted.

Figure 32 illustrates how the reference count is affected by the `_add_ref()` and `_remove_ref()` functions.

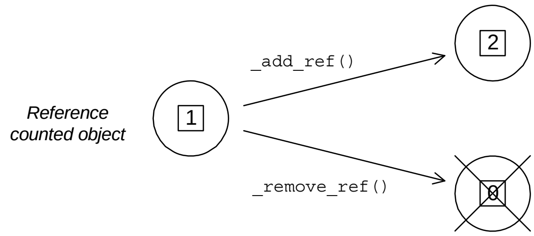

Figure 32: *Reference Counted Object*

Var template class

Table 60 shows the basic operations supported by the `IT_Bus::Var<T>` template class.

Table 60: *Basic IT_Bus::Var<T> Operations*

Operation	Description
`=`	The assignment operator distinguishes between the following kinds of assignment: • Assigning a plain pointer to a Var. • Assigning a Var to a Var.
`*`	Dereferences the Var (returning the referenced object).
`->`	Accesses the members of the referenced object.
`T* get()`	Returns a plain pointer to the referenced object. The reference count is unchanged.
`T* release()`	Returns a plain pointer to the referenced object and gives up ownership of the object (the Var resets to null). The reference count is unchanged.

Assigning a plain pointer to a Var

When a plain pointer is assigned to a Var, the Var type takes ownership of one reference count unit and leaves the reference count unchanged. For example, suppose that `Foo` is a reference counted class (that is, Foo inherits from IT_Bus::RefCountedBase). The following example shows what happens when a plain pointer to Foo, plain_p, is assigned to a Var type, fV.

```
// C++
#include <it_bus/var.h>
...
{
    Foo* plain_p = new Foo();        // Initially, ref count = 1

    // Assign the plain pointer, plain_p, to the Var, f_v
    IT_Bus::Var<Foo> f_v = plain_p; // Ref count = 1

    // f_v automatically decreases ref count to 0 at end of scope
}
```

There is no need to delete the plain_p pointer explicitly. The f_v destructor automatically reduces the reference count by 1 when it comes to the end of the current scope, resulting in the destruction of the original Foo object.

Figure 33 shows the state of the variables in the preceding example just after the assignment to the Var, f_v.

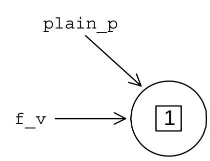

Figure 33: *After Assigning a Plain Pointer to a Var*

> **Note:** You should *never* attempt to delete a reference counted object directly. To ensure clean-up, you can either assign the reference counted object to a Var or call _remove_ref().

Assigning a Var to a Var

When a Var is assigned to a Var, the reference count is increased by one. For example, suppose that Foo is a reference counted class (that is, Foo inherits from IT_Bus::RefCountedBase). The following example shows what happens when a Var pointer, f1_v, is copied twice, into f2_v and f3_v.

```
// C++
#include <it_bus/var.h>
...
{
    IT_Bus::Var<Foo> f1_v = new Foo();  // Initially, ref count = 1

    IT_Bus::Var<Foo> f2_v = f1_v;       // Ref count = 2
    IT_Bus::Var<Foo> f3_v = f1_v;       // Ref count = 3

    // Vars automatically decrease ref count to 0 at end of scope
}
```

The use of Var types ensures that the original Foo object is deleted at the end of the current scope (because the reference count goes to 0).

Figure 34 shows the state of the variables in the preceding example just after the assignment to the Var, f3_v.

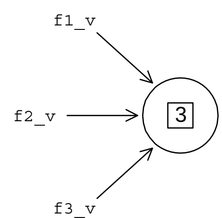

Figure 34: *A Reference Counted Object Referenced by Three Vars*

Casting from a plain pointer to a Var

To cast a plain pointer to a Var, use the standard C++ cast operators: dynamic_cast<*T*>, static_cast<*T*>, and const_cast<*T*>.

Casting from a Var to a Var

To cast a Var to a Var, Artix provides the following casting operators:

```
// C++
IT_Bus::dynamic_cast_var<T>()
IT_Bus::static_cast_var<T>()
IT_Bus::const_cast_var<T>()
```

These operate analogously to the standard C++ cast operators, dynamic_cast<*T*>, static_cast<*T*>, and const_cast<*T*>, with the additional side effect that the reference count increases by one (the casting operators call _add_ref() on the referenced object).

Examples of casting

For some examples of using the IT_Bus::dynamic_cast_var<T> operator, see “Reflection Example” on page 455.

Reflection API

This section briefly describes the Artix reflection API. The header files for the classes described in this section are located in *ArtixInstallDir*/include/it_bus/reflect.

This section contains the following subsections:

- Overview of the Reflection API
- IT_Reflect::Value<T>
- IT_Reflect::All

- IT_Reflect::Sequence
- IT_Reflect::Choice
- IT_Reflect::SimpleContent
- IT_Reflect::ComplexContent
- IT_Reflect::ElementList
- IT_Reflect::SimpleTypeList
- IT_Reflect::Nillable

Overview of the Reflection API

Artix provides a collection of reflection classes to parse the contents of XML schema data objects. Figure 35 gives an overview of the inheritance hierarchy for this C++ reflection API.

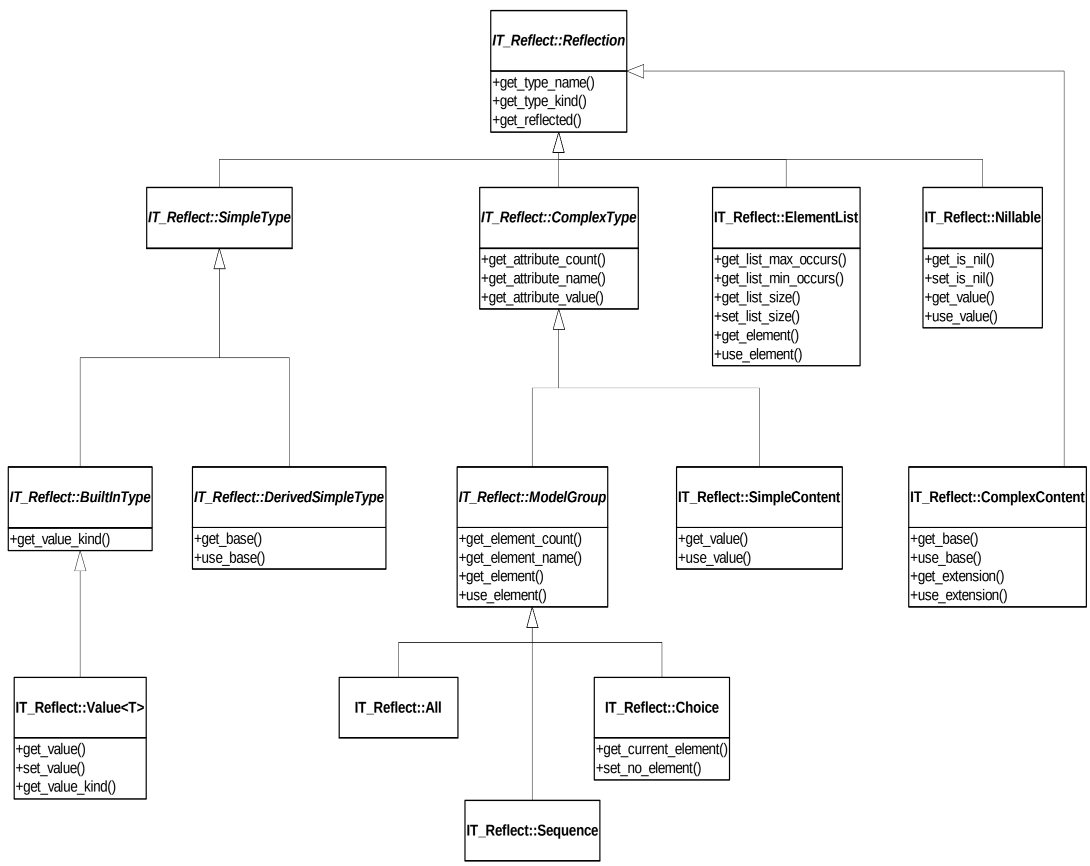

Figure 35: *Reflection API Inheritance Hierarchy*

Base classes

The following classes in Figure 35 on page 437 are used as base classes:

`IT_Reflect::Reflection`	Base class for all reflection classes.
`IT_Reflect::SimpleType`	Base class for all built-in and restricted simple types.
`IT_Reflect::BuiltInType`	Base class for all built-in types.
`IT_Reflect::ComplexType`	Base class for all complex types (types with attributes) except `ComplexContent`.
`IT_Reflect::ModelGroup`	Base class for `xsd:all`, `xsd:sequence` and `xsd:choice` types.

Leaf classes

The following classes in Figure 35 on page 437 are the leaf classes for the reflection API:

`IT_Reflect::Value<T>`	Template class for built-in types.
`IT_Reflect::DerivedSimpleType`	Reflection class for restricted simple types.
`IT_Reflect::All`	Reflection class for the `xsd:all` type.
`IT_Reflect::Sequence`	Reflection class for the `xsd:sequence` type.
`IT_Reflect::Choice`	Reflection class for the `xsd:choice` type.
`IT_Reflect::SimpleContent`	Reflection class for `xsd:simpleContent` types.
`IT_Reflect::ComplexContent`	Reflection class for `xsd:complexContent` types.
`IT_Reflect::ElementList`	Reflection class representing an element declared with non-default `minOccurs` or non-default `maxOccurs` properties.
`IT_Reflect::Nillable`	Reflection class representing an element declared with `nillable="true"`.

IT_Reflect::Value<T>

The `IT_Reflect::Value<T>` template class is used to represent built-in types.

This subsection discusses the following topics:

- Sample schema.
- IT_Reflect::Value<T> template class.

- IT_Reflect::Value<T> member functions.
- Example.

Sample schema

Example 253 shows an example of schema element defined to be of simple type, xsd:string.

Example 253:*Simple Type Example Element*

```
<schema targetNamespace="http://schemas.iona.com/example"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://schemas.iona.com/example">
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <element name="string_elem" type="xsd:string"/>
</schema>
```

IT_Reflect::Value<T> template class

The IT_Reflect::Value<*T*> template class can be used to define a reflection class for each of the standard built-in schema types. For example, you would declare IT_Reflect::Value<IT_Bus::Boolean> to hold an xsd:boolean, IT_Reflect::Value<IT_Bus::Short> to hold an xsd:short, and IT_Reflect::Value<IT_Bus::String> to hold an xsd:string.

IT_Reflect::Value<T> member functions

Example 254 shows the IT_Reflect::Value<*T*> member functions, which enable you to read and modify the value of a simple type using the get_value() and set_value() functions.

Example 254:*IT_Reflect::Value<T> Member Functions*

```
// C++

// Member functions defined in IT_Reflect::Value<T>
const T& get_value() const IT_THROW_DECL(());

T&       get_value() IT_THROW_DECL(());

void     set_value(const T& value) IT_THROW_DECL(());

IT_Reflect::BuiltInType::ValueKind
get_value_kind() const IT_THROW_DECL(());

// Member functions inherited from IT_Reflect::BuiltInType
IT_Reflect::BuiltInType::ValueKind
get_value_kind() const IT_THROW_DECL(()) = 0;

void copy(const IT_Reflect::BuiltInType* other)
   IT_THROW_DECL((IT_Reflect::ReflectException));
```

Example 254: *IT_Reflect::Value<T> Member Functions*

```
IT_Bus::Boolean equals(const IT_Reflect::BuiltInType* other)
   const IT_THROW_DECL(());

// Member functions inherited from IT_Reflect::Reflection
const IT_Bus::QName&
get_type_name() const IT_THROW_DECL(());

IT_Bus::AnyType::Kind
get_type_kind() const IT_THROW_DECL(());

const IT_Bus::AnyType&
get_reflected() const IT_THROW_DECL(());

IT_Bus::AnyType&
get_reflected() IT_THROW_DECL(());

IT_Bus::AnyType*
clone() const IT_THROW_DECL((ReflectException));
```

Identifying a built-in type

The `IT_Reflect::BuiltInType` class (base class of `IT_Reflect::Value<T>`) supports two functions that return type information, as follows:

```
//C++
IT_Bus::AnyType::Kind
get_type_kind() const IT_THROW_DECL(());

IT_Reflect::BuiltInType::ValueKind
get_value_kind() const IT_THROW_DECL(()) = 0;
```

When parsing a reflection object containing a built-in type, you can use the preceding functions as follows:

get_type_kind()

This function returns the value, `BUILT_IN`, for *all* built-in types. Hence, it can be used to determine that the reflection object is a built-in type, but it does not identify exactly which kind of built-in type.

get_value_kind()

This function tells you the precise kind of built-in type. For example, it returns `FLOAT`, if the reflection object is of `xsd:float` type, or `ANY_HOLDER`, if the reflection object is of `xsd:anyType` type.

Atomic built-in types

For a complete list of supported atomic types, see Table 31 on page 273.

Other built-in types

For the list of supported non-atomic types, see Table 61.

Table 61: *Non-Atomic Built-In Types Supported by Reflection*

Value Kind	Schema Type	C++ Type
ANYURI	xsd:anyURI	IT_Bus::AnyURI
ANY	xsd:any	IT_Bus::Any
ANY_LIST	xsd:any *(multiply occurring)*	IT_Bus::AnyList
ANY_HOLDER	xsd:anyType	IT_Bus::AnyHolder
REFERENCE	wsa:EndpointReferenceType	WS_Addressing::EndpointReferenceType

Example

You can access and modify an `xsd:string` basic type as follows:

```
// C++
IT_Reflect::Value<IT_Bus::String>& v_str = // ...

// Read the string value.
cout << "Element string value = " << v_str.get_value() << endl;

// Change the string value.
v_str.set_value("New string value here.");
```

IT_Reflect::All

The `IT_Reflect::All` reflection class represents the `xsd:all` type. This class supports functions to access an unordered group of elements and functions to access and modify attributes.

This subsection discusses the following topics:

- Sample schema.
- IT_Reflect::All member functions.

Sample schema

Example 255 shows a sample schema for an `xsd:all` type.

Example 255: *All Type Example Schema*

```
<schema targetNamespace="http://schemas.iona.com/example"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://schemas.iona.com/example">
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <complexType name="SimpleAll">
        <all>
            <element name="varFloat" type="float"/>
            <element name="varInt" type="int"/>
            <element name="varString" type="string"/>
        </all>
        <attribute name="varAttrString" type="string"/>
    </complexType>
</schema>
```

IT_Reflect::All member functions

Example 256 shows the `IT_Reflect::All` member functions, which enable you to access and modify the contents and attributes of an `xsd:all` type.

Example 256: *IT_Reflect::All Member Functions*

```
// C++
// Member functions inherited from IT_Reflect::ModelGroup
const IT_Bus::QName& get_element_name(size_t i) const IT_THROW_DECL(());

size_t get_element_count() const IT_THROW_DECL(());

IT_Bus::QName get_element_name(size_t i) const IT_THROW_DECL((ReflectException));

const IT_Reflect::Reflection*
get_element(size_t i) const IT_THROW_DECL((ReflectException));

const IT_Reflect::Reflection*
get_element(const IT_Bus::QName& element_name) const IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_element(size_t i) IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_element(
    const IT_Bus::QName& element_name
) IT_THROW_DECL((ReflectException));

// Member functions inherited from IT_Reflect::ComplexType
const IT_Bus::QName& get_attribute_name(size_t i) const IT_THROW_DECL(());

size_t get_attribute_count() const IT_THROW_DECL(());

const IT_Reflect::Reflection*
get_attribute_value(size_t i) const IT_THROW_DECL((ReflectException));
```

Example 256:*IT_Reflect::All Member Functions*

```
const IT_Reflect::Reflection*
get_attribute_value(const IT_Bus::QName& name) const IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_attribute_value(size_t i) IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_attribute_value(const IT_Bus::QName& name) IT_THROW_DECL((ReflectException));
// Member functions inherited from IT_Reflect::Reflection
const IT_Bus::QName&
get_type_name() const IT_THROW_DECL(());

IT_Bus::AnyType::Kind
get_type_kind() const IT_THROW_DECL(());

const IT_Bus::AnyType&
get_reflected() const IT_THROW_DECL(());

IT_Bus::AnyType&
get_reflected() IT_THROW_DECL(());

IT_Bus::AnyType*
clone() const IT_THROW_DECL((ReflectException));
```

IT_Reflect::Sequence

The `IT_Reflect::Sequence` reflection class represents the `xsd:sequence` type. This class supports functions to access an *ordered* group of elements and functions to access and modify attributes.

This subsection discusses the following topics:

- Sample schema.
- IT_Reflect::Sequence member functions.

Sample schema

Example 257 shows a sample schema for an `xsd:sequence` type.

Example 257: *Sequence Type Example Schema*

```
<schema targetNamespace="http://schemas.iona.com/example"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://schemas.iona.com/example">
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <complexType name="SimpleStruct">
        <sequence>
            <element name="varFloat" type="float"/>
            <element name="varInt" type="int"/>
            <element name="varString" type="string"/>
        </sequence>
        <attribute name="varAttrString" type="string"/>
    </complexType>
</schema>
```

IT_Reflect::Sequence member functions

Example 258 shows the `IT_Reflect::Sequence` member functions, which enable you to access and modify the contents and attributes of an `xsd:sequence` type.

Example 258: *IT_Reflect::Sequence Member Functions*

```
// C++
// Member functions defined in IT_Reflect::Sequence
IT_Reflect::Reflection& get_element_at(size_t index) IT_THROW_DECL((ReflectException));

const IT_Reflect::Reflection& get_element_at(size_t index) const
   IT_THROW_DECL((ReflectException));

// Member functions inherited from IT_Reflect::ModelGroup
const IT_Bus::QName& get_element_name(size_t i) const IT_THROW_DECL(());

size_t get_element_count() const IT_THROW_DECL(());

IT_Bus::QName get_element_name(size_t i) const IT_THROW_DECL((ReflectException));

const IT_Reflect::Reflection*
get_element(size_t i) const IT_THROW_DECL((ReflectException));

const IT_Reflect::Reflection*
get_element(const IT_Bus::QName& element_name) const IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_element(size_t i) IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_element(
    const IT_Bus::QName& element_name
) IT_THROW_DECL((ReflectException));

// Member functions inherited from IT_Reflect::ComplexType
```

Example 258: *IT_Reflect::Sequence Member Functions*

```
const IT_Bus::QName& get_attribute_name(size_t i) const IT_THROW_DECL(());

size_t get_attribute_count() const IT_THROW_DECL(());

const IT_Reflect::Reflection*
get_attribute_value(size_t i) const IT_THROW_DECL((ReflectException));

const IT_Reflect::Reflection*
get_attribute_value(const IT_Bus::QName& name) const IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_attribute_value(size_t i) IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_attribute_value(const IT_Bus::QName& name) IT_THROW_DECL((ReflectException));

// Member functions inherited from IT_Reflect::Reflection
const IT_Bus::QName&
get_type_name() const IT_THROW_DECL(());

IT_Bus::AnyType::Kind
get_type_kind() const IT_THROW_DECL(());

const IT_Bus::AnyType&
get_reflected() const IT_THROW_DECL(());

IT_Bus::AnyType&
get_reflected() IT_THROW_DECL(());

IT_Bus::AnyType*
clone() const IT_THROW_DECL((ReflectException));
```

IT_Reflect::Choice

The `IT_Reflect::Choice` reflection class represents the `xsd:choice` type. This class supports functions to access the choice element and functions to access and modify attributes.

This subsection discusses the following topics:

- Sample schema.
- IT_Reflect::Choice member functions.

Sample schema

Example 259 shows a sample schema for an `xsd:choice` type.

Example 259: *Choice Type Example Schema*

```
<schema targetNamespace="http://schemas.iona.com/example"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://schemas.iona.com/example">
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

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

IT_Reflect::Choice member functions

Example 260 shows the `IT_Reflect::Choice` member functions, which enable you to access and modify the contents and attributes of an `xsd:choice` type.

Example 260: *IT_Reflect::Choice Member Functions*

```
// C++
// Member functions defined in IT_Reflect::Choice
IT_Bus::QName
get_element_name() const IT_THROW_DECL(());

// Member functions inherited from IT_Reflect::ModelGroup
const IT_Bus::QName& get_element_name(size_t i) const IT_THROW_DECL(());

size_t get_element_count() const IT_THROW_DECL(());

IT_Bus::QName get_element_name(size_t i) const IT_THROW_DECL((ReflectException));

const IT_Reflect::Reflection*
get_element(size_t i) const IT_THROW_DECL((ReflectException));

const IT_Reflect::Reflection*
get_element(const IT_Bus::QName& element_name) const IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_element(size_t i) IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_element(
    const IT_Bus::QName& element_name
) IT_THROW_DECL((ReflectException));

// Member functions inherited from IT_Reflect::ComplexType
const IT_Bus::QName& get_attribute_name(size_t i) const IT_THROW_DECL(());

size_t get_attribute_count() const IT_THROW_DECL(());
```

Example 260:*IT_Reflect::Choice Member Functions*

```
const IT_Reflect::Reflection*
get_attribute_value(size_t i) const IT_THROW_DECL((ReflectException));

const IT_Reflect::Reflection*
get_attribute_value(const IT_Bus::QName& name) const IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_attribute_value(size_t i) IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_attribute_value(const IT_Bus::QName& name) IT_THROW_DECL((ReflectException));

// Member functions inherited from IT_Reflect::Reflection
const IT_Bus::QName&
get_type_name() const IT_THROW_DECL(());

IT_Bus::AnyType::Kind
get_type_kind() const IT_THROW_DECL(());

const IT_Bus::AnyType&
get_reflected() const IT_THROW_DECL(());

IT_Bus::AnyType&
get_reflected() IT_THROW_DECL(());

IT_Bus::AnyType*
clone() const IT_THROW_DECL((ReflectException));
```

IT_Reflect::SimpleContent

The `IT_Reflect::SimpleContent` reflection class represents types defined using the `<xsd:simpleContent>` tag. This class supports functions to access the type's value and functions to access and modify attributes. Simple content types can be derived either by restriction or by extension from existing simple types (see "Deriving a Complex Type from a Simple Type" on page 324 for more details).

This subsection discusses the following topics:

- Sample schema.
- IT_Reflect::SimpleContent member functions.

Sample schema

Example 261 shows a sample schema for an `xsd:simpleContent` type.

Example 261: *SimpleContent Type Example Schema*

```
<schema targetNamespace="http://schemas.iona.com/example"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://schemas.iona.com/example">
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

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

IT_Reflect::SimpleContent member functions

Example 262 shows the `IT_Reflect::SimpleContent` member functions, which enable you to access and modify the contents and attributes of an `xsd:simpleContent` type.

Example 262: *IT_Reflect::SimpleContent Member Functions*

```
// C++
// Member functions defined in IT_Reflect::SimpleContent
IT_Reflect::Reflection*
use_value() IT_THROW_DECL(());

const IT_Reflect::Reflection*
get_value() const IT_THROW_DECL(());

// Member functions inherited from IT_Reflect::ComplexType
const IT_Bus::QName& get_attribute_name(size_t i) const IT_THROW_DECL(());

size_t get_attribute_count() const IT_THROW_DECL(());

const IT_Reflect::Reflection*
get_attribute_value(size_t i) const IT_THROW_DECL((ReflectException));

const IT_Reflect::Reflection*
get_attribute_value(const IT_Bus::QName& name) const IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_attribute_value(size_t i) IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_attribute_value(const IT_Bus::QName& name) IT_THROW_DECL((ReflectException));

// Member functions inherited from IT_Reflect::Reflection
const IT_Bus::QName&
```

Example 262: *IT_Reflect::SimpleContent Member Functions*

```
get_type_name() const IT_THROW_DECL(());

IT_Bus::AnyType::Kind
get_type_kind() const IT_THROW_DECL(());

const IT_Bus::AnyType&
get_reflected() const IT_THROW_DECL(());

IT_Bus::AnyType&
get_reflected() IT_THROW_DECL(());

IT_Bus::AnyType*
clone() const IT_THROW_DECL((ReflectException));
```

IT_Reflect::ComplexContent

The `IT_Reflect::ComplexContent` reflection class represents types defined using the `<xsd:complexContent>` tag. This class supports functions to access the type's base contents and derived contents, as well as functions to access and modify attributes. Complex content types can be derived by extension from existing types (see "Deriving a Complex Type from a Complex Type" on page 326 for more details).

This subsection discusses the following topics:

- Sample schema.
- IT_Reflect::ComplexContent member functions.

Sample schema

Example 263 shows a sample schema for an `xsd:complexContent` type.

Example 263: *ComplexContent Type Example Schema*

```
<schema targetNamespace="http://schemas.iona.com/example"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://schemas.iona.com/example">
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <complexContent mixed="false">
        <extension base="tns:SimpleStruct">
            <sequence>
                <element name="varStringExt" type="string"/>
                <element name="varFloatExt" type="float"/>
            </sequence>
            <attribute name="attrString1" type="string"/>
        </extension>
    </complexContent>
</schema>
```

IT_Reflect::ComplexContent member functions

Example 264 shows the `IT_Reflect::SimpleContent` member functions, which enable you to access and modify the contents and attributes of an `xsd:complexContent` type.

Example 264:*IT_Reflect::ComplexContent Member Functions*

```
// C++
// Member functions defined in IT_Reflect::ComplexContent
const IT_Reflect::Reflection*
get_base() const IT_THROW_DECL((IT_Reflect::ReflectException));

IT_Reflect::Reflection*
use_base() IT_THROW_DECL((IT_Reflect::ReflectException));

const IT_Reflect::Reflection* get_extension() const
   IT_THROW_DECL((IT_Reflect::ReflectException));

IT_Reflect::Reflection*
use_extension() IT_THROW_DECL((IT_Reflect::ReflectException));

// Member functions inherited from IT_Reflect::ComplexType
const IT_Bus::QName& get_attribute_name(size_t i) const IT_THROW_DECL(());

size_t get_attribute_count() const IT_THROW_DECL(());

const IT_Reflect::Reflection*
get_attribute_value(size_t i) const IT_THROW_DECL((ReflectException));

const IT_Reflect::Reflection*
get_attribute_value(const IT_Bus::QName& name) const IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_attribute_value(size_t i) IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_attribute_value(const IT_Bus::QName& name) IT_THROW_DECL((ReflectException));

// Member functions inherited from IT_Reflect::Reflection
const IT_Bus::QName&
get_type_name() const IT_THROW_DECL(());

IT_Bus::AnyType::Kind
get_type_kind() const IT_THROW_DECL(());

const IT_Bus::AnyType&
get_reflected() const IT_THROW_DECL(());

IT_Bus::AnyType&
get_reflected() IT_THROW_DECL(());

IT_Bus::AnyType*
clone() const IT_THROW_DECL((ReflectException));
```

Testing for an extension

If the complex content data type does not have an extension part, the `get_extension()` and `use_extension()` functions return 0 (NULL pointer).

IT_Reflect::ElementList

The `IT_Reflect::ElementList` reflection class represents an element declared with non-default `minOccurs` or non-default `maxOccurs` properties. Specifically, if you call a reflection function that accesses an element, there are two possible return values from that function, depending on the values of `minOccurs` and `maxOccurs`:

`minOccurs="1"` `maxOccurs="1"`	Returns the element directly.
All other values	Returns `IT_Reflect::ElementList`.

It makes no difference whether `minOccurs` and `maxOccurs` are set explicitly or get their values by default.

This subsection discusses the following topics:

- Sample schema.
- IT_Reflect::ElementList member functions.

Sample schema

Example 265 shows a sample schema for an Artix array, which is represented as an element list.

Example 265:*Artix Array Type Example Schema*

```
<schema targetNamespace="http://schemas.iona.com/example"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://schemas.iona.com/example">
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <complexType name="ArrayOfString">
        <sequence>
            <element name="varString" type="xsd:string"
                     minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
    </complexType>
</schema>
```

IT_Reflect::ElementList member functions

Example 266 shows the `IT_Reflect::ElementList` member functions, which enable you to access and modify the contents of an Artix array type.

Example 266:*IT_Reflect::ElementList Member Functions*

```
// C++
// Member functions defined in IT_Reflect::ElementList
size_t get_list_max_occurs() const IT_THROW_DECL(());

size_t get_list_min_occurs() const IT_THROW_DECL(());

size_t get_list_size() const IT_THROW_DECL(());

void set_list_size(size_t size) IT_THROW_DECL((ReflectException));

const IT_Reflect::Reflection*
get_element(size_t index) const IT_THROW_DECL((ReflectException));

IT_Reflect::Reflection*
use_element(size_t index) IT_THROW_DECL((ReflectException));

// Member functions defined in IT_Reflect::Reflection
const IT_Bus::QName&
get_type_name() const IT_THROW_DECL(());

IT_Bus::AnyType::Kind
get_type_kind() const IT_THROW_DECL(());

const IT_Bus::AnyType&
get_reflected() const IT_THROW_DECL(());

IT_Bus::AnyType&
get_reflected() IT_THROW_DECL(());

IT_Bus::AnyType*
clone() const IT_THROW_DECL((ReflectException));
```

IT_Reflect::SimpleTypeList

The IT_Reflect::SimpleTypeList class is fairly similar to the IT_Reflect::ElementList class, except that the values in the list are restricted to be of IT_Bus::SimpleType type. The elements of an IT_Reflect::SimpleTypeList instance are accessed using the following functions:

Example 267: *get_item() and use_item() Functions from SimpleTypeList*

```
// C++
const IT_Bus::SimpleType*
get_item(
    size_t index
) const IT_THROW_DECL((IT_Reflect::ReflectException)) = 0;

IT_Bus::SimpleType*
use_item(
    size_t index
) IT_THROW_DECL((IT_Reflect::ReflectException)) = 0;
```

IT_Reflect::Nillable

The IT_Reflect::Nillable reflection class represents an element declared with nillable="true". Specifically, if you call a reflection function that accesses an element, the return values from that function, depend on the value of nillable and on the values of minOccurs and maxOccurs, as follows:

Table 62: *Effect of nillable, minOccurs and maxOccurs Settings*

nillable	minOccurs / maxOccurs	Return Value
nillable="false"	minOccurs="1" maxOccurs="1"	Returns the element directly.
nillable="false"	*All other values*	Returns IT_Reflect::ElementList.
nillable="true"	minOccurs="1" maxOccurs="1"	Returns IT_Reflect::Nillable containing an element directly.
nillable="true"	*All other values*	Returns an IT_Reflect::ElementList containing a list of IT_Reflect::Nillables.

It makes no difference whether minOccurs and maxOccurs are set explicitly or get their values by default.

This subsection discusses the following topics:

- Sample schema.
- IT_Reflect::Nillable member functions.

Sample schema

Example 268 shows a sample schema for a sequence type with nillable elements.

Example 268:*Sequence Type with Nillable Elements Example Schema*

```
<schema targetNamespace="http://schemas.iona.com/example"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://schemas.iona.com/example">
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <complexType name="StructWithNillables">
        <sequence>
            <element name="varFloat" nillable="true"
                     type="float"/>
            <element name="varInt" nillable="true" type="int"/>
            <element name="varString" nillable="true"
                     type="string"/>
            <element name="varStruct" nillable="true"
                     type="tns:SimpleStruct"/>
        </sequence>
    </complexType>
</schema>
```

IT_Reflect::Nillable member functions

Example 269 shows the `IT_Reflect::Nillable` member functions, which enable you to access and modify the contents of a nillable type.

Example 269:*IT_Reflect::Nillable Member Functions*

```
// C++
// Member functions defined in IT_Reflect::Nillable
IT_Bus::Boolean get_is_nil() const IT_THROW_DECL(());

void set_is_nil() IT_THROW_DECL(());

const IT_Reflect::Reflection*
get_value() const IT_THROW_DECL((IT_Reflect::ReflectException));

IT_Reflect::Reflection*
use_value() IT_THROW_DECL((ReflectException));

// Member functions defined in IT_Reflect::Reflection
const IT_Bus::QName&
get_type_name() const IT_THROW_DECL(());

IT_Bus::AnyType::Kind
get_type_kind() const IT_THROW_DECL(());

const IT_Bus::AnyType&
get_reflected() const IT_THROW_DECL(());

IT_Bus::AnyType&
get_reflected() IT_THROW_DECL(());
```

Example 269: *IT_Reflect::Nillable Member Functions*

```
IT_Bus::AnyType*
clone() const IT_THROW_DECL((ReflectException));
```

Reflection Example

As an example of Artix reflection, this section describes a program that is capable of printing the contents of any Artix data type (including built-in and user-defined types). The code examples in this section are taken from the `print_random` demonstration.

This section contains the following subsections:

Print an IT_Bus::AnyType

This subsection describes the main `print()` function for the `Printer` class, which has the following signature:

```
void Printer::print(const IT_Bus::AnyType& any);
```

This function enables you to print out any XML type in Artix, including built-in types and user-defined types (for built-in types, you have to insert the data into an `IT_Bus::AnyHolder` instance before calling `print()`). All user-defined types and the `IT_Bus::AnyHolder` type derive from `IT_Bus::AnyType`.

The `print(const IT_Bus::AnyType&)` function immediately calls `IT_Bus::AnyType::get_reflection()` to convert the `AnyType` to an `IT_Reflect::Reflection` instance. Parsing and printing of the `Reflection` instance is then performed by the `print(const IT_Reflect::Reflection*)` function.

Code extract

Example 270 shows a code extract from the `Printer` class, which shows the top-level functions for printing an `IT_Bus::AnyType` instance using the Artix Reflection API.

Example 270: *Code Example for Printing an IT_Bus::AnyType Instance*

```
// C++
#include "printer.h"
#include <it_bus/any_type.h>
#include <it_bus/reflect/complex_content.h>
#include <it_bus/reflect/complex_type.h>
#include <it_bus/reflect/element_list.h>
#include <it_bus/reflect/choice.h>
#include <it_bus/reflect/nillable.h>
```

Example 270: *Code Example for Printing an IT_Bus::AnyType Instance*

```
#include <it_bus/reflect/reflection.h>
#include <it_bus/reflect/simple_content.h>
#include <it_bus/reflect/simple_type.h>
#include <it_bus/reflect/derived_simple_type.h>
#include <it_bus/reflect/built_in_type.h>
#include <it_bus/reflect/value.h>
#include <it_cal/iostream.h>
IT_USING_NAMESPACE_STD;
using namespace IT_Bus;

1 class Indenter
{
  public:
    Indenter(Printer* p) : m_p(p) { m_p->indent(); }
    ~Indenter() { m_p->outdent(); }
  private:
    Printer* m_p;
};

IT_ostream&
Printer::start_line()
{
    for (int i = 0; i < m_indent; ++i)
    {
        cout << "    ";
    }
    return cout;
}

void
Printer::indent()
{
    m_indent++;
}

void
Printer::outdent()
{
    m_indent--;
}


void
2 Printer::print(
    const AnyType& any,
    int indent
)
{
3   Var<const IT_Reflect::Reflection> reflection(any.get_reflection());
    Printer printer;
    printer.m_indent = indent;
4   printer.print(reflection.get());
}

Printer::Printer()
    :
    m_indent(0),
    m_in_list(IT_FALSE)
```

Example 270: *Code Example for Printing an IT_Bus::AnyType Instance*

```
{
}

Printer::~Printer()
{
}

void
5 Printer::print(
    const IT_Reflect::Reflection* reflection
)
{
    assert(reflection != 0);
6   switch (reflection->get_type_kind())
    {
      case AnyType::BUILT_IN:
        print(IT_DYNAMIC_CAST(const IT_Reflect::BuiltInType*, reflection));
        break;
      case AnyType::SIMPLE:
7       print(IT_DYNAMIC_CAST(const IT_Reflect::DerivedSimpleType*,
   reflection));
        break;
      case AnyType::SEQUENCE:
      case AnyType::ALL:
        print(IT_DYNAMIC_CAST(const IT_Reflect::ModelGroup*, reflection));
        break;
      case AnyType::CHOICE:
        print(IT_DYNAMIC_CAST(const IT_Reflect::Choice*, reflection));
        break;
      case AnyType::SIMPLE_CONTENT:
        print(IT_DYNAMIC_CAST(const IT_Reflect::SimpleContent*, reflection));
        break;
      case AnyType::ELEMENT_LIST:
        print(IT_DYNAMIC_CAST(const IT_Reflect::ElementList*, reflection));
        break;
      case AnyType::COMPLEX_CONTENT:
        print(IT_DYNAMIC_CAST(const IT_Reflect::ComplexContent*, reflection));
        break;
      case AnyType::NILLABLE:
        print(IT_DYNAMIC_CAST(const IT_Reflect::Nillable*, reflection));
        break;

      default:
        String message(
            "<Unsupported type: "+reflection->get_type_name().to_string()+">");
        throw Exception(message);
    }
}
```

The preceding extract from the Printer class implementation can be explained as follows:

1. The Indenter class, together with the Printer::start_line(), Printer::indent(), and Printer::outdent() functions, are used by the Printer class to produce the output in a neatly indented format.
2. The Printer::print(const IT_Bus::AnyType&) function is a static member function that prints XML schema data types that inherit from xsd:anyType (effectively, any XML type). This print() function is the most important function exposed by the Printer class and you can use it to print any XML type, irrespective of whether stub code for the type is available or not.
3. The IT_Bus::AnyType instance, any, is converted to an IT_Reflect::Reflection instance by calling get_reflection(). The IT_Bus::Var<*T*> template type is just a reference counting smart pointer type. See "The IT_Bus::Var Template Type" on page 433 for more details.
4. The reflection.get() call returns a pointer of const IT_Reflect::Reflection* type, which can then be passed as the argument to Printer::print(const IT_Reflect::Reflection*).
5. The Printer::print(const IT_Reflect::Reflection*) function is the root print function for printing reflection instances. This print function recursively iterates over the contents of the reflection instance, printing all of its data.
6. The switch statement determines structure of the reflection object, based on its type. The IT_Reflect::Reflection::get_type_kind() function returns an enumeration of IT_Bus::AnyType::Kind type.
7. Cast the IT_Reflection::Reflection object to the appropriate type, based on its kind. The IT_DYNAMIC_CAST(A,B) preprocessor macro is equivalent to a conventional C++ dynamic_cast<*T*> operator.

Print Atomic and Simple Types

This subsection describes the print() functions for printing XML simple types. These functions have the following signatures:

```
void Printer::print(const IT_Reflect::BuiltInType*);
void Printer::print(const IT_Reflect::DerivedSimpleType*);
```

The IT_Reflect::SimpleType class is the base class for all simple types and the following classes derive from SimpleType:

- IT_Reflect::BuiltInType—the base class for the IT_Reflect::Value<*T*> types that reflect an XML built-in type. For example, the IT_Reflect::Value<IT_Bus::Int> reflection type derives from BuiltInType.
- IT_Reflect::DerivedSimpleType—the class that reflects simple types derived by restriction from built-in types.

This example makes extensive use of C++ templates to simplify the processing of all the different XML built-in types.

Code extract

Example 271 shows a code extract from the `Printer` class, which shows the functions for printing XML atomic and simple types using the Artix Reflection API.

Example 271:*Code Example for Printing Atomic and Simple Types*

```
   // C++
   template <class T>
   void
1  print_atom(
       const T& value
   )
   {
       cout << value << endl;
   }

   template <>
   void
2  print_atom(
       const QName& value
   )
   {
       cout << value.to_string() << endl;
   }


   /** A template to print value reflections values. */
   template <class T>
   struct PrintValue
   {
       static void
3      print_value(
           const IT_Reflect::SimpleType* data,
           Printer& printer
       )
       {
           if (printer.is_in_list())
           {
               printer.start_line();
           }
4          const IT_Reflect::Value<T>* value =
               IT_DYNAMIC_CAST(const IT_Reflect::Value<T>*, data);
           assert(value != 0);
           print_atom(value->get_value());
       }
   };


   void
5  Printer::print(
       const IT_Reflect::DerivedSimpleType* data
   )
   {
       assert(data != 0);
6      Var<const IT_Reflect::SimpleType> base(data->get_base());
       print(base.get());
       return;
```

Example 271: *Code Example for Printing Atomic and Simple Types*

```
}

void
7 Printer::print(
    const IT_Reflect::BuiltInType* data
)
{
    assert(data != 0);
8   switch (data->get_value_kind())
    {
      case IT_Reflect::BuiltInType::BOOLEAN:
9       PrintValue<Boolean>::print_value(data, *this);
        return;
      case IT_Reflect::BuiltInType::FLOAT:
        PrintValue<Float>::print_value(data, *this);
        return;
      case IT_Reflect::BuiltInType::DOUBLE:
        PrintValue<Double>::print_value(data, *this);
        return;
      case IT_Reflect::BuiltInType::INT:
        PrintValue<Int>::print_value(data, *this);
        return;
      case IT_Reflect::BuiltInType::LONG:
        PrintValue<Long>::print_value(data, *this);
        return;
      case IT_Reflect::BuiltInType::SHORT:
        PrintValue<Short>::print_value(data, *this);
        return;
      case IT_Reflect::BuiltInType::UINT:
        PrintValue<UInt>::print_value(data, *this);
        return;
      case IT_Reflect::BuiltInType::ULONG:
        PrintValue<ULong>::print_value(data, *this);
        return;
      case IT_Reflect::BuiltInType::USHORT:
        PrintValue<UShort>::print_value(data, *this);
        return;
      case IT_Reflect::BuiltInType::BYTE:
        PrintValue<Byte>::print_value(data, *this);
        return;
      case IT_Reflect::BuiltInType::UBYTE:
        PrintValue<UByte>::print_value(data, *this);
        return;
      case IT_Reflect::BuiltInType::STRING:
        PrintValue<String>::print_value(data, *this);
        return;
      case IT_Reflect::BuiltInType::DECIMAL:
        PrintValue<Decimal>::print_value(data, *this);
        return;
      case IT_Reflect::BuiltInType::QNAME:
        PrintValue<QName>::print_value(data, *this);
        return;

        // Other types not implemented in this demo
      case IT_Reflect::BuiltInType::HEXBINARY:
      case IT_Reflect::BuiltInType::BASE64BINARY:
      case IT_Reflect::BuiltInType::DATE:
      case IT_Reflect::BuiltInType::TIME:
```

Example 271: *Code Example for Printing Atomic and Simple Types*

```
        case IT_Reflect::BuiltInType::ANYURI:
        case IT_Reflect::BuiltInType::ID:
        case IT_Reflect::BuiltInType::DATETIME:
        case IT_Reflect::BuiltInType::ANY:
        case IT_Reflect::BuiltInType::ANY_LIST:
        case IT_Reflect::BuiltInType::ANY_HOLDER:
        case IT_Reflect::BuiltInType::REFERENCE:
        default:
          start_line() << "not implemented:" << data->get_type_name().to_string()
                       << endl;
    }
}
```

The preceding extract from the `Printer` class implementation can be explained as follows:

1. The `print_atom<T>()` function template is a template for printing out most simple types, such as `IT_Bus::Boolean`, `IT_Bus::Int`, and so on.
2. The `print_atom<IT_Bus::QName>` function is a specialization of the `print_atom<T>` template for printing qualified names, of `IT_Bus::QName` type.
3. The `PrintValue<T>::print_value()` function template is a simple wrapper function that combines a dynamic type cast with a call to `print_atomic<T>()`.
4. The `IT_DYNAMIC_CAST(A,B)` preprocessor macro is equivalent to a conventional C++ `dynamic_cast<T>` operator.
5. The `Printer::print(const IT_Reflect::DerivedSimpleType*)` function prints derived simple types. See “Deriving Simple Types by Restriction” on page 299 for details of a simple type derived by restriction.
6. This line accesses the value of the derived simple type by calling the `IT_Bus::DerivedSimpleType::get_base()` function.
7. The `Printer::print(const IT_Reflect::BuiltInType*)` function prints out all of the XML built-in types.
8. The `IT_Reflect::BuiltInType::get_value_kind()` function returns an enumeration of `IT_Reflect::BuiltInType::ValueKind` type.
9. The built-in types can be printed using the appropriate form of the `PrintValue<T>::print_value()` template function.

Print Sequence, Choice, and All Types

This subsection describes the `print()` functions for printing XML sequence, choice and all types (collectively known as the *model group* types in the XML syntax).

The `print()` function for sequence and all types has the following signature:

```
void Printer::print(const IT_Reflect::ModelGroup*);
```

The `print()` function for choice types has the following signature:

```
void Printer::print(const IT_Reflect::Choice*);
```

Code extract for sequence and all

Example 272 shows a code extract from the `Printer` class, which shows the functions for printing XML sequence and all types using the Artix Reflection API.

Example 272:*Code Example for Printing Sequence and All Types*

```
  // C++
  void
1 Printer::print(
      const IT_Reflect::ModelGroup* data
  )
  {
      assert(data != 0);
      cout << endl;
      start_line();
2     switch (data->get_type_kind())
      {
        case AnyType::SEQUENCE: cout << "Sequence "; break;
        case AnyType::ALL: cout << "All "; break;
        default: assert(0);
      }
3     cout << data->get_type_name().to_string() << ": " << endl;
4     print_attributes(data);
      start_line() << "Value" << endl;
      Indenter indent(this);
5     for (int i = 0; i < data->get_element_count(); ++i)
      {
6         Var<const IT_Reflect::Reflection>
              element(data->get_element(i));
7         start_line() << data->get_element_name(i).to_string() << ": ";
          Indenter indent(this);
8         print(element.get());
      }
  }
```

The preceding extract from the `Printer` class implementation can be explained as follows:

1. The `Printer::print(const IT_Reflect::ModelGroup*)` function prints reflection instances that represent sequence or all types.
2. The `IT_Reflect::Reflection::get_type_kind()` function returns an enumeration of `IT_Bus::AnyType::Kind` type.
3. The `IT_Reflect::Reflection::get_type_name()` function returns the QName of the current type. The `IT_Bus::QName` type is converted to a string using the `to_string()` function.
4. The attributes for this instance are printed out by calling the `Printer::print_attributes(const IT_Reflect::ComplexType*)` function. See "Print ComplexContent Types" on page 464 for a description of this function.
5. Iterate over all the elements in the sequence or all.
6. The `Var<const IT_Reflect::Reflection>` type is used to construct a reference counted smart pointer to an element instance, `element`. See "The IT_Bus::Var Template Type" on page 433 for details.

7. The get_element_name() function returns a QName, which is converted to a string using the to_string() function.
8. This line passes the element object to the generic reflection print function, Printer::print(const IT_Reflect::Reflection*).

Code extract for choice

Example 273 shows a code extract from the Printer class, which shows the function for printing XML choice types using the Artix Reflection API.

Example 273: *Code Example for Printing Choice Types*

```
// C++
void
1 Printer::print(
    const IT_Reflect::Choice* data
)
{
    assert(data != 0);
    cout << endl;
2   start_line() << "Choice "
                 << data->get_type_name().to_string() << endl;
    Indenter indent(this);
    print_attributes(data);
    start_line() << "Value:" << endl;
    Indenter indent2(this);
3   int i = data->get_current_element();
    if (i != -1)
    {
        Var<const IT_Reflect::Reflection>
                                     element(data->get_element(i));
        start_line() << data->get_element_name(i).to_string()
                     << ": ";
        Indenter indent3(this);
4       print(element.get());
    }
}
```

The preceding extract from the Printer class implementation can be explained as follows:

1. The Printer::print(const IT_Reflect::Choice*) function prints reflection instances that represent choice types.
2. The IT_Reflect::Reflection::get_type_name() function returns the QName of the current type.
3. The IT_Reflect::Choice::get_current_element() function returns the index of the current element (or -1 if no element is selected).
4. The get() function converts the IT_Bus::Var<*T*> smart pointer into a plain pointer—see “The IT_Bus::Var Template Type” on page 433. In this case, the returned pointer is of IT_Reflect::Reflection* type.

Print SimpleContent Types

This subsection describes the print() function for printing XML simple content types (defined using the <xsd:simpleContent> tag). The simple content print() function has the following signature:

```
void Printer::print(const IT_Reflect::SimpleContent*);
```

A simple content type is an XML schema complex type that can have attributes, but contains no sub-elements.

Code extract

Example 274 shows a code extract from the Printer class, which shows the function for printing XML schema xsd:simpleContent types using the Artix reflection API.

Example 274: *Code Example for Printing SimpleContent Types*

```
   // C++
   void
1  Printer::print(
       const IT_Reflect::SimpleContent* data
   )
   {
       assert(data != 0);
       cout << endl;
       start_line() << "simpleContentComplexType "
                    << data->get_type_name().to_string() << ": " << endl;
2      print_attributes(data);
       start_line() << "Value: " << endl;
       Indenter indent(this);
3      Var<const IT_Reflect::SimpleType> value(data->get_value());
       print(value.get());
   }
```

The preceding extract from the Printer class implementation can be explained as follows:

1. The Printer::print(const IT_Reflect::SimpleContent*) function prints reflection instances that represent simple content types (that is, complex types that can have attributes, but no subelements).
2. The attributes for this instance are printed out by calling the Printer::print_attributes(const IT_Reflect::ComplexType*) function. See “Print ComplexContent Types” on page 464 for a description of this function.
3. The Var<const IT_Reflect::SimpleType> type is a reference counting smart pointer. The value variable references the contents of the SimpleContents type.

Print ComplexContent Types

This subsection describes the print() function for printing XML complex content types (defined using the <xsd:complexContent> tag). The complex content print() function has the following signature:

```
void Printer::print(const IT_Reflect::ComplexContent*);
```

A complex content type can have attributes, can contain sub-elements and can be used to define complex types that derive from other complex types (see “Deriving a Complex Type from a Complex Type” on page 326).

Code extract

Example 275 shows a code extract from the `Printer` class, which shows the functions for printing XML schema `xsd:complexContent` types using the Artix reflection API.

Example 275:*Code Example for Printing ComplexContent Types*

```
  // C++
  void
1 Printer::print(
      const IT_Reflect::ComplexContent* data
  )
  {
      assert(data != 0);
      cout << endl;
2     start_line() << "complexContentComplexType "
                   << data->get_type_name().to_string() << ": "
                   << endl;
3     Var<const IT_Reflect::Reflection> base(data->get_base());
      start_line() << "Base part: " << endl;
      {
          Indenter indent(this);
          print(base.get());
      }
4     Var<const IT_Reflect::Reflection>
                  extension(data->get_extension());
      if (extension.get())
      {
          start_line() << "Extension part: " << endl;
          Indenter indent(this);
          print(extension.get());
      }
  }

  void
5 Printer::print_attributes(
      const IT_Reflect::ComplexType* data
  )
  {
      assert(data != 0);
      start_line() << "Attributes: " << endl;
      Indenter indent(this);
6     for (size_t i = 0; i < data->get_attribute_count(); ++i)
      {
7         Var<const IT_Reflect::Reflection> value(
              data->get_attribute_value(
                  data->get_attribute_name(i)
              )
          );
          start_line() << data->get_attribute_name(i).to_string()
                       << " = ";
          if (value.get() == 0)
```

Example 275: *Code Example for Printing ComplexContent Types*

```
        {
            cout << "<missing>" << endl;
        }
        else
        {
            print(value.get());
        }
    }
    assert(data != 0);
}
```

The preceding extract from the `Printer` class implementation can be explained as follows:

1. The `Printer::print(const IT_Reflect::ComplexContent*)` function prints XML schema `xsd:complexContent` types (that is, complex types that can have attributes *and* subelements).
2. The `IT_Reflect::Reflection::get_type_name()` function returns the QName of the current complex content type.
3. Construct a `Var<const IT_Reflect::Reflection>` smart pointer type to reference the base contents of the `xsd:complexContent` type. The base contents will be non-empty, if the `xsd:complexContent` type is defined by derivation—see "Deriving a Complex Type from a Complex Type" on page 326 for details.
4. Construct a `Var<const IT_Reflect::Reflection>` smart pointer type to reference the extended (that is, derived) contents of the `xsd:complexContent` type.
5. The `Printer::print_attributes(const IT_Reflect::ComplexType*)` function prints out the list of attributes for any complex type.
6. Iterate over all of the attributes associated with this element.
7. If an attribute is defined with `use="optional"` in the XML schema, for example:

   ```
   <attribute name="AttrName" type="AttrType" use="optional"/>
   ```

   Then the value returned from the `get_attribute_value()` function could be a NULL pointer (that is, `0`), if the attribute is not set.

Print Multiple Occurrences

This subsection describes the `print()` function for printing element lists (objects of `IT_Reflect::ElementList` type). The `print()` function for a multiply-occurring element has the following signature:

```
void Printer::print(const IT_Reflect::ElementList*);
```

An `IT_Reflect::ElementList` object is used to represent elements defined with non-default values of `minOccurs` and `maxOccurs` (that is, any values apart from `minOccus=1` and `maxOccurs=1`). Calling a `get_element()` function can return an `IT_Reflect::ElementList` object instead of a single element, if the element is multiply occurring.

Code extract

Example 276 shows a code extract from the `Printer` class, which shows the function for printing multiply occurring elements (represented by the `IT_Reflect::ElementList` type) using the Artix reflection API.

Example 276: *Code Example for Printing Multiple Occurrences*

```
   // C++
   void
1  Printer::print(
       const IT_Reflect::ElementList* data
   )
   {
       assert(data != 0);
       m_in_list = true;
       cout << endl;
2      for (size_t i = 0; i < data->get_list_size(); ++i)
       {
3          Var<const IT_Reflect::Reflection>
               element(data->get_element(i));
           print(element.get());
       }
       m_in_list = false;
   }

   bool
   Printer::is_in_list()
   {
       return m_in_list;
   }
```

The preceding extract from the `Printer` class implementation can be explained as follows:

1. The `Printer::print(const IT_Reflect::ElementList*)` function prints multiply occurring elements (that is, elements whose occurrence constraints have any values except the defaults, `minOccurs="1"` and `maxOccurs="2"`).
2. The `IT_Reflect::ElementList::get_size()` function returns the number of elements in the element list.
3. Construct a `Var<const IT_Reflect::Reflection>` smart pointer type to reference the `i`th element in the list.

Print Nillables

This subsection describes the `print()` function for printing nillable elements (objects of `IT_Reflect::Nillable` type). The `print()` function for a nillable element has the following signature:

```
void Printer::print(const IT_Reflect::Nillable*);
```

An `IT_Reflect::Nillable` object is used to represent elements defined with `nillable="true"`. In this case, the value of the element might be absent (`IT_Reflect::Nillable::is_nil()` equals `true`). If the element is non-nil, it can be retrieved by calling `IT_Reflect::Nillable::get_value()`.

Code extract

Example 277 shows a code extract from the `Printer` class, which shows the function for printing nillables using the Artix reflection API.

Example 277:*Code Example for Printing Nillables*

```
  // C++
  void
1 Printer::print(
      const IT_Reflect::Nillable* data
  )
  {
      assert(data != 0);
2     if (data->get_is_nil())
      {
          cout << "<nil>" << endl;
      }
      else
      {
3         Var<const IT_Reflect::Reflection>
              value(data->get_value());
          print(value.get());
      }
  }
```

The preceding extract from the `Printer` class implementation can be explained as follows:

1. The `Printer::print(const IT_Reflect::Nillable*)` function prints nillable elements (that is, elements defined with the attribute `xsd:nillable="true"` in the XML schema).
2. Test the nillable element for nilness using the `IT_Reflect::Nillable::is_nil()` function before attempting to print the element value.
3. Construct a `Var<const IT_Reflect::Reflection>` smart pointer type to reference the value of the nillable.

Persistent Maps

Artix provides a persistence mechanism, built on top of Berkeley DB, which you can use to persist your Artix data types. You must use this persistence mechanism, if you intend to integrate your application with Artix high availability (HA).

Introduction to Persistent Maps

Artix *persistent maps* constitute a simple persistence mechanism, which is tailored to work with Artix data types and is based on Berkeley DB.

The persistent map API is concerned solely with inserting and extracting records to and from a persistent map. The details of setting up the Berkeley DB are taken care of by configuration—see "Configuration Example" on page 481. Once you have configured your application to use Berkeley DB, a new Berkeley DB instance is automatically created when you start the application for the first time. No programming is required in order to create the database or to connect to the database.

Header files

The following header file is always needed for the persistent map API:

```
it_bus_pdk/persistent_map.h
```

The following header files might also be needed, depending on your persistence requirements:

```
it_bus_pdk/persistent_string_map.h
it_bus_pdk/qname_persistence_handler.h
it_bus_pdk/any_type_persistence_handler.h
```

DBConfig type

An instance of `IT_Bus::DBConfig` type encapsulates all of the Berkeley DB configuration details. Implicitly, when a `DBConfig` instance is created, it reads the configuration details from the application's configuration scope (in the `artix.cfg` configuration file).

You do not need to call any of the `DBConfig` member functions. A `DBConfig` instance is needed only for passing to a persistent map constructor.

Persistent map templates

The persistent map templates are used to construct hash tables that are stored persistently in the Berkeley database. The hash table stores pairs of items: the first item is a *key*, which can be of arbitrary type, and the second item is *data*, which can also be of arbitrary type.

The following persistent map templates are provided:

- `IT_Bus::PersistentStringMap<>` template

  A hash table that uses `IT_String` (which can implicitly convert to and from `IT_Bus::String`) for the key and any atomic type (for example, `char` or `int`) for the data. To use this type, you must include the `it_bus_pdk/persistent_string_map.h` header.
- `IT_Bus::PersistentMap<>` template

  A hash table that uses any atomic (for example, `char` or `int`) type for the key and any atomic type for the data.
- `IT_Bus::PersistentMapBase<>` template

  A hash table that uses any type (atomic or complex) for the key and any type (atomic or complex) for the data.

Persistence handler types

The persistence handler types are used internally by Artix to make data persistent. You do not need to use persistence handler types directly; you provide them as template arguments to the `PersistentMapBase` template.

The following handler types are provided:

- `IT_Bus::PODPersistenceHandler`

  Used by Artix to make simple atomic types (such as `char`, `int` and so on) persistent.
- `IT_Bus::StringPersistenceHandler`

  Used by Artix to make the `IT_String` type (or `IT_Bus::String` type) persistent. To use this type, you must include the `it_bus_pdk/persistent_string_map.h` header.
- `IT_Bus::QNamePersistenceHandler`

  Used by Artix to make the `IT_Bus::QName` type persistent. To use this type, you must include the `it_bus_pdk/qname_persistence_handler.h` header.
- `IT_Bus::AnyTypePersistenceHandler<>` template

  Used by Artix to make complex types persistent. Specifically, the `AnyTypePersistenceHandler` can persist any type that inherits from `IT_Bus::AnyType`, which includes any complex types generated from a WSDL contract or an XML schema.

  To use this type, you must include the `it_bus_pdk/any_type_persistence_handler.h` header and link with the `it_bus_xml` library.

Creating a Persistent Map

This section describes how to create persistent maps using the `PersistentStringMap<>`, `PersistentMap<>`, and `PersistentMapBase<>` templates.

Persistent map constructor

In general, the constructor for a *PersistentMapType* persistent map has the following signature:

```
// C++
PersistentMapType::PersistentMapType(
    const char* id,
    DBConfig*   cfg
};
```

The constructor takes the following arguments:

- `id`—a unique string that identifies the persistent map instance in the database. You can choose any string for the `id`, as long as it does not clash with a pre-existing perstent map instance.
- `cfg`—a pointer to an `IT_Bus::DBConfig` instance.

Lifetime of DBConfig instance

You can access the Berkeley DB only as long as the `DBConfig` instance continues to exist. Therefore, you must avoid deleting this object prematurely. Typically, you would create a `DBConfig` instance near the beginning of your application's `main()` function (just after initializing an `IT_Bus::Bus` instance) and destroy the DBConfig instance near the end of the `main()` function.

Creating a persistent string map

An `IT_Bus::PersistentStringMap<>` template is a persistent map type that uses an `IT_String` type or an `IT_Bus::String` type as its key and any atomic type (such as `char` or `int`) as its data.

Example 278 shows you how to create a string persistent map, `f_map`, that uses `float` as its data type.

Example 278:*Creating a String Persistent Map*

```
// C++
using namespace IT_Bus;

typedef IT_Bus::PersistentStringMap<float> FloatMap;
DBConfig cfg(bus);         // bus is an initialized bus instance
FloatMap f_map("StringToFloat", &cfg);
```

Creating a persistent map for atomic types

An `IT_Bus::PersistentMap<>` template is a persistent map type that uses any atomic type as its key and any atomic type as its data.

Example 279 shows you how to create a persistent map, `i_map`, that uses `char` as its key type and `int` as its data type.

Example 279:*Creating a Persistent Map for Atomic Types*

```
// C++
using namespace IT_Bus;

typedef IT_Bus::PersistentMap<char, int> IntMap;
DBConfig cfg(bus);         // bus is an initialized bus instance
IntMap i_map("CharToInt", &cfg);
```

Creating a persistent map for complex types

To create a persistent map type, *PersistentMapType*, for complex data, define a `typedef` of the `IT_Bus::PersistentMapBase<>` template as follows:

```
// C++
typedef IT_Bus::PersistentMapBase<
        KeyType,
        DataType,
        KeyPersistenceHandler,
        DataPersistenceHandler
    > PersistentMapType;
```

Where both the *KeyType* and the *DataType* types can either be a atomic type (`char`, `int` and so on) or a complex type. The *KeyPersistenceHandler* and *DataPersistenceHandler* types must be chosen to match the corresponding *KeyType* and *DataType* types. See “Persistence handler types” on page 470 for the complete list of persistence handler types.

Example 280 shows you how to create two persistent maps using the `PersistentMapBase` template: the `QtoRMap` type maps QNames to `WS_Addressing::EndpointReferenceType` instances and the `CharToWSDLMap` type maps `char`s to instances of a user complex type, `MyWSDLType`.

Example 280: *Creating a Persistent Map for Complex Types*

```
// C++
using namespace IT_Bus;

typedef IT_Bus::PersistentMapBase<
      IT_Bus::QName,
      WS_Addressing::EndpointReferenceType,
      IT_Bus::QNamePersistenceHandler,

   IT_Bus::AnyTypePersistenceHandler<WS_Addressing::EndpointReferenceType>
    > QtoRMap;

typedef IT_Bus::PersistentMapBase<
      char,
      MyWSDLType,
      IT_Bus::PODPersistenceHandler,
      IT_Bus::AnyTypePersistenceHandler<MyWSDLType>
    > ChartoWSDLMap;
DBConfig cfg(bus);
QtoRMap map("myRefMap", &cfg);
ChartoWSDLMap myMap("myDataMap", &cfg);
```

Inserting, Extracting, and Removing Data

This section explains how to perform basic operations on persistent maps. The following tasks are described here:

- Inserting data into a persistent map.
- Extracting data from a persistent map.
- Removing data from a persistent map.
- Avoiding deadlock with iterators.

Inserting data into a persistent map

To insert data into a persistent map of *PersistentMapType* type, perform the following steps:

1. Create a *PersistentMapType*`::value_type` object to hold the (key, data) pair.
2. Insert the value type into the map using the *PersistentMapType*`::insert()` function.

If `insert()` succeeds, the data is committed right away to the database. The operation is an atomic transaction and you do not have control over the transactionality of the operation.

Example of a simple insert

Given a persistent map instance, i_map, of IntMap type (see Example 279 on page 472), you can insert a (key, data) pair as follows:

```
// C++
IntMap::value_type val('a', 175);
i_map.insert(val);
```

Example of an insert with overwriting

The insert() function takes a second optional parameter that determines whether to over-write an existing record in the persistent map. A value of true implies the data is over-written, if the key matches an existing record; a value of false (the default) implies the data is not over-written.

Given a persistent map instance, i_map, of IntMap type, you can over-write a (key, data) pair as follows:

```
// C++
IntMap::value_type val('a', 190);
i_map.insert(val, true);
```

Example of an insert with error checking

The insert() function returns an IT_Pair containing an *PersistentMapType*::iterator and an IT_Bool. Hence, you can optionally define a pair object of IT_Pair<*PersistentMapType*::iterator, IT_Bool> type to hold the return value from a *PersistentMapType*::insert() call.

If the insert succeeds in writing to the database, the returned iterator, pair.first, is a valid pointer to the inserted record and the returned boolean, pair.second, is true. If the insert *cannot* write the record (for example, a record was already present and you did not specify overwriting) the iterator points to the existing record and the boolean is false.

Given a persistent map instance, i_map, of IntMap type, you can check whether a value insertions succeeds, as follows:

```
// C++
IntMap::value_type val('a', 200);
IT_Pair<IntMap::iterator, IT_Bool> pair;
pair = i_map.insert(val);
if (!pair.second)
{
    // handle the error
}
```

Extracting data from a persistent map

To retrieve data from a persistent database, call the *PersistentMapType*::find() function, passing the key value of the record you want to access. For example, if a persistent map consists of (char, int) pairs, the find() function takes a char argument.

The `find()` function returns a *PersistentMapType*`::iterator` object, which is effectively a pointer to an `IT_Pair` object. Using the iterator, you can view the value of the desired record and also iterate through the remaining entries in the database. Unlike iterators for in-memory hash maps, however, you cannot alter the values in the database using this iterator.

Example of extracting data

To find a record keyed by the `char` value, `'a'`, from a persistent map, `i_map`, of `IntMap` type, call `find()` as follows:

```
// C++

// Restrict the scope of the iterator object
{
    IntMap::iterator iter = i_map.find('a');
    if (iter != i_map.end()) {
        // prints out the value of the int stored with key 'a'
        cout << (*iter).second << endl;
    }
}
```

WARNING: An iterator object holds a lock on the Berkeley DB and this lock is not released until the iterator is destroyed. Hence, to avoid deadlock, it is essential to delete the iterator object (or let it go out of scope) before making any further calls that require a lock, such as `insert()` or `erase()`.

Removing data from a persistent map

To remove a record from a persistent map, call the *PersistentMapType*`::erase()` function, passing the key value of the record you want to erase as the sole argument. Like `insert()`, the `erase()` function is atomic: if it succeeds, the data on the disk is updated right away.

Example of removing a record

To erase a record keyed by the `char` value, `'a'`, from a persistent map, `i_map`, of `IntMap` type, call `erase()` as follows:

```
// C++
// Removes the record with key 'a'
if ( i_map.erase('a') ) {
    cout << "Record successfully erased!" << endl;
}
```

Avoiding deadlock with iterators

Persistent map iterators are implemented using Berkeley DB cursors, which acquire a read lock on the underlying database, and this lock is held until the iterator is destroyed. It follows that you cannot perform any locking operations (such as `insert()` or `erase()`) as long as an iterator object exists for the persistent map.

The following example shows an *incorrect* code fragment using iterators that leads to deadlock:

```
// C++
IntMap::iterator iter = i_map.find('a');
if (iter == i_map.end())
{
    IntMap::value_type val('a', 123);
    i_map.insert(val);  // DEADLOCK!
}
```

The correct way to implement this code is as follows:

```
// C++
bool found = false;
{
    IntMap::iterator iter = i_map.find('a');
    found = (iter != i_map.end());
}
if (!found)
{
    IntMap::value_type val('a', 123);
    i_map.insert(val);  // No deadlock, iterator is gone.
}
```

Handling Exceptions

Artix provides a specific type, `IT_Bus::DBException`, to represent the database exceptions thrown by functions from the persistent map API. Database exceptions should typically be handled on the server side (for example, by writing the exception message to a server-side log).

Exception handling sample

Example 281 shows how Artix database exceptions should be handled on the server side for applications that use the persistent map API.

Example 281:*Sample Operation with DB Exception Handling*

```
   // C++
   #include <it_bus_pdk/db_exception.h>

   void
1  foo() IT_THROW_DECL((IT_Bus::Exception))
   {
       try
       {
           // Catch and process DBException explicitly
           m_persistent_map.find(...);
           ...
       }
2      catch (const IT_Bus::DBException& db_ex)
```

Example 281: *Sample Operation with DB Exception Handling*

```
    {
3       // Handle DB error locally...
        ...
    }
}
```

The preceding exception handling sample can be explained as follows:

1. In this example, `foo()` represents the implementation of a WSDL operation (in other words, it is a member function of a servant class).
2. Persistent map operations can throw exceptions of `IT_Bus::DBException` type, which inherits from the generic Artix exception class, `IT_Bus::Exception`.
3. The DB exceptions should be handled locally, on the server side.

IT_Bus::DBException class

Example 282 shows the signatures of the member functions from the `IT_Bus::DBException` class.

Example 282: *The IT_Bus::DBException Class*

```
// C++
namespace IT_Bus {
    class IT_BUS_API DBException :
        public Exception,
        public Rethrowable<DBException>
    {
      public:
        DBException(
            unsigned long exception_type,
            int                  native_error_code,
            const char*          msg
        );
        DBException(const DBException& rhs);
        virtual ~DBException();

        IT_ULong    error() const;
        const char* error_as_string() const;
        const char* message() const;
        int         native_error_code() const;
        ...
    };
}
```

The `DBException` class exposes the following member functions:

- `error()`

  Returns an Artix database error code (see “Database minor exception codes” on page 478). The code returned from this function is usually the most convenient way to distinguish the type of error that occurred.
- `error_as_string()`

Returns the name of an Artix database error code.

- `message()`

  Returns a descriptive error message string, which you could use for writing the error to a log.

- `native_error_code()`

  Returns a native Berkeley DB error code.

Database minor exception codes

The following minor exception codes can be returned by the `IT_Bus::DBException::error()` function.

Example 283:*Database Exception Error Codes*

```
// C++
// DBException error() codes.
IT_Bus::DB_EXCEPTION_CANNOT_WRITE_LOCK_FILE
IT_Bus::DB_EXCEPTION_FAILURE_DURING_GET
IT_Bus::DB_EXCEPTION_FAILURE_DURING_PUT
IT_Bus::DB_EXCEPTION_FAILURE_DURING_ERASE
IT_Bus::DB_EXCEPTION_FAILURE_DURING_GET_SIZE
IT_Bus::DB_EXCEPTION_COULD_NOT_CREATE_SHARED_DB_ENV
IT_Bus::DB_EXCEPTION_COULD_NOT_OPEN_SHARED_DB_ENV
IT_Bus::DB_EXCEPTION_COULD_NOT_CREATE_DB
IT_Bus::DB_EXCEPTION_COULD_NOT_OPEN_DB
IT_Bus::DB_EXCEPTION_NULL_POINTER
IT_Bus::DB_EXCEPTION_COULD_NOT_CREATE_CURSOR
IT_Bus::DB_EXCEPTION_COULD_NOT_DUP_CURSOR
IT_Bus::DB_EXCEPTION_FAILURE_DURING_GET_VALUE
IT_Bus::DB_EXCEPTION_COULD_NOT_INITIALIZE_REPLICATION
IT_Bus::DB_EXCEPTION_COULD_NOT_INIT_TXN
IT_Bus::DB_EXCEPTION_COULD_NOT_COMMIT_TXN
IT_Bus::DB_EXCEPTION_COULD_NOT_MKDIR_DB_HOME
IT_Bus::DB_EXCEPTION_BAD_CONFIGURATION
IT_Bus::DB_EXCEPTION_COULD_NOT_OPEN_SYNC_DB
IT_Bus::DB_EXCEPTION_COULD_NOT_CREATE_SYNC_DB
IT_Bus::DB_EXCEPTION_COULD_NOT_WRITE_TO_SYNC_DB
IT_Bus::DB_EXCEPTION_SYNC_DB_NOT_READY
IT_Bus::DB_EXCEPTION_COULD_NOT_PROMOTE
IT_Bus::DB_EXCEPTION_COULD_NOT_DEMOTE
IT_Bus::DB_EXCEPTION_SLAVE_CANNOT_UPDATE_DB
IT_Bus::DB_EXCEPTION_LICENSE_CHECK_FAILED
IT_Bus::DB_EXCEPTION_ENV_IN_USE
```

Supporting High Availability

If you are going to use persistent maps in conjunction with the high availability features of Artix, it is necessary to perform some additional programming tasks to support *write-request forwarding*. Essentially, you must write a few lines of code to tell Artix which WSDL operations need to write to the database (using the persistent map API).

> **Note:** The write-request forwarding feature is currently not supported by the CORBA binding.

Write-request forwarding

The high availability model in Artix mirrors the high availability features of the Berkeley DB. In this model, a replicated cluster consists of a *master replica* and any number of *slave replicas*. The master replica can perform both read and write operations to the database, but the slaves can perform only read operations.

What happens, though, if a client sends a write-request to one of the slave replicas? In this case, the slave replica needs to have some way of forwarding the write-request to the master replica. Artix supports this write-request forwarding feature using the `request_forwarder` plug-in on the server side. To enable the write-request forwarding feature, you must appropriately configure the server replicas, as described in *Configuring and Deploying Artix Solutions*, and you must perform some programming steps, as described here.

Write-request forwarding API

The `IT_Bus::DBConfig` class provides the following member function to support write-request forwarding:

```
// C++
void
mark_as_write_operations(
    IT_Vector<IT_Bus::String>   operations,
    const IT_Bus::QName&        service,
    const IT_Bus::String&       port,
    const IT_Bus::String&       wsdl_url
) IT_THROW_DECL((DBException));
```

After creating a `DBConfig` instance on the server side, you should call this function to identify those WSDL operations that require a database write. The `mark_as_write_operations()` function takes the following parameters:

- `operations`—the list of WSDL operation names that require a database write (the names in this list are unqualified).
- `service`—the QName of the service whose operations are considered for forwarding.
- `port`—the name of the port whose operations are considered for forwarding.

- wsdl_url—the location of the WSDL contract.

Example code

Example 284 is an example that shows you how to program write-request forwarding. In this example, the add_employee and remove_employee operations are designated as write operations.

Example 284: *Write-Request Forwarding Example*

```
// C++
using namespace IT_Bus;

// Typical Artix server mainline
1 QName service("", "SOAPService",
   "http://www.iona.com/hello_world_soap_http");
String port_name  = "Server2";
String wsdl_url   = "hello_world.wsdl";
Bus_var bus = IT_Bus::init(...);
DBConfig db_cfg(bus);

2 IT_Vector<String> write_operations;
write_operations.push_back("add_employee");
write_operations.push_back("remove_employee");

3 db_cfg.mark_as_write_operations(
    write_operations,
    service,
    port_name,
    wsdl_url
);

// Now register servant as normal
4 bus->register_servant(
        servant,
        wsdl_url,
        service,
        port_name
);
```

The preceding code can be described as follows:

1. The service, service, and port, port_name, defined here are used to identify the port whose operations are considered for forwarding.
2. The list of write operations is constructed as a vector of strings, IT_Vector<IT_Bus::String>, which is similar to the std::vector type from the standard template library (see "IT_Vector Template Class" on page 408).
3. Call the IT_Bus::DBConfig::mark_as_write_operations() function to set the write operations from the given service and port, which are considered for forwarding.
4. The servant registered by this line of code is the one whose operations are considered for forwarding. The service and port name arguments used here are identical to the service and port name arguments passed to the mark_as_write_operations() function.

High availability demonstration

A demonstration that illustrates the Artix high availability functionality is available at the following location:

ArtixInstallDir/cxx_java/samples/advanced/high_availability_persistent_servers

Configuration Example

Example 285 shows the minimal configuration that is required to configure persistence based on the Berkeley DB.

Example 285:*Configuration Required for Using Berkeley DB in Artix*

```
# Artix Configuration File
...
foo_service {
    plugins:artix:db:env_name = "myDB.env";
    plugins:artix:db:home = "/etc/dbs/foo_service";
};
```

The following configuration variables must be set:

`plugins:artix:db:env_name`	Specifies the filename for the Berkeley DB environment file. It can be any string and can have any file extension (for example, `myDB.env`).
`plugins:artix:db:home`	Specifies the directory where Berkeley DB stores all the files for the service databases. Each service should have a dedicated folder for its data stores. This is especially important for replicated services.

Reference

For more details about how to configure persistence, particularly for configuring high availability features, see the relevant chapter on high availability in ***Configuring and Deploying Artix Solutions***.

WSDL-to-C++ Compiler Utility

Use the wsdltocpp compiler utility to generate C++ stub code, starting point code and makefiles from a WSDL contract. The Artix WSDL-to-C++ mapping conforms to the official OMG specification, http://www.omg.org/cgi-bin/doc?mars/06-06-38.

Generating Stubs and Starting Point Code

The Artix development tools include a utility to generate server skeleton and client stub code from an Artix contract. The generated code has the following features:

- Artix generated code is compatible with a multitude of transports.
- Artix maps WSDL types to C++ using a proprietary WSDL-to-C++ mapping.

Generated files

The Artix code generator produces a number of stub files from the Artix contract. They are named according to the port type name, *PortTypeName*, specified in the logical portion of the Artix contract. If the contract specifies more than one port type, code will be generated for each one.

The following stub files are generated:

*PortTypeName***.h** defines the superclass from which the client and server are implemented. It represents the API used by the service defined in the contract.

*PortTypeName***Service.h and** *PortTypeName***Service.cxx** are the server-side skeleton code to implement the service defined in the contract.

*PortTypeName***Client.h and** *PortTypeName***Client.cxx** are the client-side stubs for implementing a client to use the service defined by the contract.

*PortTypeName***_wsdlTypes.h and** *PortTypeName***_wsdlTypes.cxx** define the complex datatypes defined in the contract (if any).

*PortTypeName***_wsdlTypesFactory.h and** *PortTypeName***_wsdlTypesFactory.cxx** define factory classes for the complex datatypes defined in the contract (if any).

Generating code from the command line

You can generate code at the command line using the command:

```
wsdltocpp [options] { WSDL-URL | SCHEMA-URL }
    [-e web_service_name[:port_list]] [-b binding_name]
    [-i port_type]* [-d output-dir] [-n URI=C++namespace]*
    [-nexclude URI[=C++namespace]]* [-ninclude URI[=C++namespace]]*
    [-nimport C++namespace] [-impl]
    [-m {NMAKE | UNIX}:[executable|library]] [-libv version]
    [ -jp plugin_class] [-f] [-server] [-client] [-sample]
    [-plugin[:plugin_name]] [-deployable] [-global] [-v]
    [-license] [-declspec declspec] [-all] [-?] [-flags]
    [-upper|-lower|-minimal|-mapper class] [-verbose] [-reflect]
```

You must specify the location of a valid WSDL contract file, *WSDL_URL*, for the code generator to work. You can also supply the following optional parameters:

`-i` *port_type*	Specifies the name of the port type for which the tool will generate code. The default is to use the first port type listed in the contract. This switch can appear multiple times.
`-e` *web_service_name* [:*port_list*]	Specifies the name of the service for which the tool will generate code. The default is to use the first service listed in the contract. You can optionally specify a comma separated list of port names to activate. The default is to activate all of the service's ports.
`-b` *binding_name*	Specifies the name of the binding to use when generating code. The default is the first binding listed in the contract.
`-d` *output_dir*	Specifies the directory to which the generated code is written. The default is the current working directory.
`-n` [*URI*=]*C++namespace*	Maps an XML namespace to a C++ namespace. The C++ stub code generated from the XML namespace, *URI*, is put into the specified C++ namespace, *C++namespace*. This switch can appear multiple times.
`-nexclude` *URI[=C++namespace]*	Do not generate C++ stub code for the specified XML namespace, *URI*. You can optionally map the XML namespace, *URI*, to a C++ namespace, *C++namespace*, in case it is referenced by the rest of the XML schema/WSDL contract. This switch can appear multiple times.
`-ninclude` *URI[=C++namespace]*	Generates C++ stub code for the specified XML namespace, *URI*. You can optionally map the XML namespace, *URI*, to a C++ namespace, *C++namespace*. This switch can appear multiple times.

`-nimport C++namespace`	Specifies the C++ namespace to use for the code generated from imported schema.
`-impl`	Generates the skeleton code for implementing the server defined by the contract.
`-m {NMAKE | UNIX}:[executable | library]`	Used in combination with `-impl` to generate a makefile for the specified platform (`NMAKE` for Windows or `UNIX` for UNIX). You can specify that the generated makefile builds an executable, by appending `:executable`, or a library, by appending `:library`. For example, the options, `-impl -m NMAKE:executable`, would generate a Windows makefile to build an executable.
`-libv version`	Used in combination with either `-m NAME:library` or `-m UNIX:library` to specify the version number of the library built by the makefile. This version number is for your own convenience, to help you keep track of your own library versions.
`-f`	*Deprecated*—No longer used (was needed to support routing in earlier versions.
`-server`	Generates stub code for a server (cannot be combined with the `-client` switch).
`-client`	Generates stub code for a client (cannot be combined with the `-server` switch).
`-sample`	Generates code for a sample implementation of a client and a server, as follows: client stub code, server stub code, a client main function and a server main function. To generate a complete working sample application, combine `-sample` with the `-impl` and the `-m` switches.
`-plugin [:plugin_name]`	Generates a service plug-in. You can optionally specify the plug-in name by appending `:plugin_name` to this option. If no plug-in name is specified, the default name is *<ServiceName><PortTypeName>*. The service name, *<ServiceName>*, is specified by the `-e` option.
`-deployable`	(Used with `-plugin`.) Generates a deployment descriptor file, `deploy<ServiceName>.xml`, which is needed to deploy a plug-in into the Artix container.

`-global`	(Used with `-plugin`.) In the generated plug-in code, instantiate the plug-in using a `GlobalBusORBPlugIn` object instead of a `BusORBPlugIn` object. A `GlobalBusORBPlugIn` initializes the plug-in automatically, as soon as it is constructed (suitable approach for plug-ins that are linked directly with application code). A `BusORBPlugIn` is not initialized unless the plug-in is either listed in the `orb_plugins` list or deployed into an Artix container (suitable approach for dynamically loading plug-ins).
`-v`	Displays the version of the tool.
`-license`	Displays the currently available licenses.
`-declspec` *`declspec`*	Creates Visual C++ declaration specifiers for `dllexport` and `dllimport`. This option makes it easier to package Artix stubs in a DLL library. See "Building Artix Stub Libraries on Windows" on page 117 for details.
`-all`	Generate stub code for all of the port types and the types that they use. This option is useful when multiple port types are defined in a WSDL contract.
`-?`	Displays help on using the command line tool.
`-flags`	Displays detailed information about the options.
`-verbose`	Send extra diagnostic messages to the console while `wsdltocpp` is running.
`-reflect`	Enables reflection on generated data classes. See "Reflection" on page 431 for details.
`-wrapped`	When used with document/literal wrapped style, generates function signatures with wrapped parameters, instead of unwrapping into separate parameters. See "Document/Literal Wrapped Style" on page 84 for details.

Note: When you generate code from WSDL that has multiple ports, multiple services, multiple bindings, or multiple port types, without specifying which port, service, binding, or port type to generate code for, the WSDL-to-C++ compiler prints a warning to the effect that it is only generating code for the first one encountered.

Index

H

I

K

L

M

N

O

P

Q

R

S

X