



Artix™

Developing Artix Applications in C++

Version 3.0, October 2005

IONA Technologies PLC and/or its subsidiaries may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this publication. Except as expressly provided in any written license agreement from IONA Technologies PLC, the furnishing of this publication does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Any rights not expressly granted herein are reserved.

IONA, IONA Technologies, the IONA logo, Orbix, Orbix Mainframe, Orbix Connect, Artix, Artix Mainframe, Artix Mainframe Developer, Mobile Orchestrator, Orbix/E, Orbacus, Enterprise Integrator, Adaptive Runtime Technology, and Making Software Work Together are trademarks or registered trademarks of IONA Technologies PLC and/or its subsidiaries.

Java and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the United States and other countries. All other trademarks that appear herein are the property of their respective owners.

While the information in this publication is believed to be accurate, IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

COPYRIGHT NOTICE

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of IONA Technologies PLC. No third-party intellectual property right liability is assumed with respect to the use of the information contained herein. IONA Technologies PLC assumes no responsibility for errors or omissions contained in this publication. This publication and features described herein are subject to change without notice.

Copyright © 2003–2005 IONA Technologies PLC. All rights reserved.

All products or services mentioned in this publication are covered by the trademarks, service marks, or product names as designated by the companies that market those products.

Updated: 29-Oct-2005

Contents

List of Tables	ix
Preface	xi
What is Covered in this Book	xi
Who Should Read this Book	xi
Finding Your Way Around the Library	xi
Searching the Artix Library	xiii
Online Help	xiii
Additional Resources	xiii
Document Conventions	xiv
Chapter 1 Developing Artix Enabled Clients and Servers	1
Generating Stub and Skeleton Code	2
C++ Namespaces	7
Defining a WSDL Interface	8
Developing a Server	10
Developing a Client	14
Generating a Sample Application from WSDL	19
Compiling and Linking an Artix Application	24
Building Artix Stub Libraries on Windows	26
Chapter 2 Artix Programming Considerations	27
Bootstrapping Service	28
How Clients Find Initial References	29
How Servers Find WSDL Contracts	31
Operations and Parameters	34
RPC/Literal Style	35
Document/Literal Wrapped Style	39
Exceptions	44
Built-In Exceptions	45
User-Defined Exceptions	47
Memory Management	51
Managing Parameters	52

Assignment and Copying	57
Deallocating	59
Smart Pointers	60
Registering Servants	64
Registering a Static Servant	65
Registering a Transient Servant	72
Multi-Threading	80
Client Threading Issues	81
Servant Threading Models	83
Setting the Servant Threading Model	86
Thread Pool Configuration	89
Converting with <code>to_string()</code> and <code>from_string()</code>	92
Locating Services with UDDI	98
Overriding a HTTP Address in a Client	100
Chapter 3 Artix References	103
Introduction to References	104
The WSDL Publish Plug-In	106
References to Transient Services	113
Programming with References	116
Bank WSDL Contract	117
Creating References	126
Resolving References	130
Chapter 4 Callbacks	131
Overview of Artix Callbacks	132
Routing and Callbacks	134
Callback WSDL Contract	138
Client Implementation	141
Server Implementation	145
Chapter 5 The Artix Locator	149
Overview of the Locator	150
Locator WSDL	153
Registering Endpoints with the Locator	159
Reading a Reference from the Locator	161

Chapter 6 Using Sessions in Artix	165
Introduction to Session Management in Artix	166
Registering a Server with the Session Manager	169
Working with Sessions	172
Chapter 7 Artix Contexts	179
Introduction to Contexts	180
Configuration Contexts	181
Header Contexts	184
Registering Contexts	186
Pre-Registered Contexts	192
Reading and Writing Context Data	196
Getting a Context Instance	197
Reading and Writing Basic Types	201
Reading and Writing User-Defined Types	203
Reading and Writing Custom Types	205
Durability of Context Settings	208
Configuration Context Example	209
HTTP-Conf Schema	210
Setting a Configuration Context on the Client Side	214
Setting a Configuration Context on the Server Side	217
Header Context Example	220
Custom SOAP Header Demonstration	221
SOAP Header Context Schema	223
Declaring the SOAP Header Explicitly	226
Client Main Function	229
Server Main Function	234
Service Implementation	237
Header Contexts in Three-Tier Systems	240
Chapter 8 Artix Data Types	243
Including and Importing Schema Definitions	244
Simple Types	246
Atomic Types	247
String Type	249
NormalizedString and Token Types	254
QName Type	259
Date and Time Types	261

Decimal Type	263
Integer Types	265
Binary Types	268
Deriving Simple Types by Restriction	275
List Type	278
Union Type	280
Holder Types	285
Unsupported Simple Types	287
Complex Types	288
Sequence Complex Types	289
Choice Complex Types	292
All Complex Types	296
Attributes	299
Attribute Groups	303
Nesting Complex Types	306
Deriving a Complex Type from a Simple Type	310
Deriving a Complex Type from a Complex Type	313
Arrays	323
Model Group Definitions	328
Wildcarding Types	333
anyURI Type	334
anyType Type	336
any Type	341
Occurrence Constraints	349
Element Occurrence Constraints	350
Sequence Occurrence Constraints	355
Choice Occurrence Constraints	359
Any Occurrence Constraints	363
Nillable Types	368
Introduction to Nillable Types	369
Nillable Atomic Types	371
Nillable User-Defined Types	375
Nested Atomic Type Nillable Elements	378
Nested User-Defined Nillable Elements	382
Nillable Elements of an Array	387
Substitution Groups	390
SOAP Arrays	400
Introduction to SOAP Arrays	401
Multi-Dimensional Arrays	405

Sparse Arrays	408
Partially Transmitted Arrays	411
IT_Vector Template Class	412
Introduction to IT_Vector	413
Summary of IT_Vector Operations	416
Unsupported XML Schema Constructs in Artix	419
Chapter 9 Artix IDL to C++ Mapping	421
Introduction to IDL Mapping	422
IDL Basic Type Mapping	424
IDL Complex Type Mapping	426
IDL Module and Interface Mapping	435
Chapter 10 Reflection	441
Introduction to Reflection	442
The IT_Bus::Var Template Type	445
Reflection API	449
Overview of the Reflection API	450
IT_Reflect::Value<T>	452
IT_Reflect::All	456
IT_Reflect::Sequence	459
IT_Reflect::Choice	462
IT_Reflect::SimpleContent	465
IT_Reflect::ComplexContent	467
IT_Reflect::ElementList	470
IT_Reflect::SimpleTypeList	472
IT_Reflect::Nillable	473
Reflection Example	476
Print an IT_Bus::AnyType	477
Print Atomic and Simple Types	482
Print Sequence, Choice and All Types	488
Print SimpleContent Types	491
Print ComplexContent Types	493
Print Multiple Occurrences	496
Print Nillables	498
Chapter 11 Persistent Maps	501
Introduction to Persistent Maps	502

Creating a Persistent Map	505
Inserting, Extracting and Removing Data	508
Handling Exceptions	512
Supporting High Availability	515
Configuration Example	518
Chapter 12 Transactions in Artix	519
Introduction to Transactions	520
Selecting the Transaction System	525
Configuring OTS Lite	526
Configuring OTS Encina	529
Configuring WS-AT	533
Transaction API	536
Transaction Demarcation	539
Participants and Resources	542
Transaction Participants	543
Interposition	550
Threading	552
Transaction Propagation	556
Notification Handlers	560
Reliable Messaging with MQ Transactions	562
Client Example	571
Appendix A http-conf Context Data Types	575
Appendix B MQ-Series Context Data Types	585
Index	601

List of Tables

Table 1: Artix Import Libraries for Linking with an Application	24
Table 2: Simple Schema Type to Simple Bus Type Mapping	247
Table 3: IANA Character Set Names	250
Table 4: Description of token and Types Derived from token	254
Table 5: Validity Testing Functions for Normalized Strings and Tokens	257
Table 6: Member Fields of <code>IT_Bus::DateTime</code>	261
Table 7: Member Fields Supported by Other Date and Time Types	262
Table 8: Operators Supported by <code>IT_Bus::Decimal</code>	263
Table 9: Unlimited Precision Integer Types	265
Table 10: Operators Supported by the Integer Types	265
Table 11: Schema to Bus Mapping for the Binary Types	268
Table 12: List of Artix Holder Types	286
Table 13: Nillable Atomic Types	371
Table 14: Member Functions Not Defined in <code>IT_Vector</code>	413
Table 15: Member Types Defined in <code>IT_Vector<T></code>	416
Table 16: Iterator Member Functions of <code>IT_Vector<T></code>	417
Table 17: Element Access Operations for <code>IT_Vector<T></code>	417
Table 18: Stack Operations for <code>IT_Vector<T></code>	417
Table 19: List Operations for <code>IT_Vector<T></code>	418
Table 20: Other Operations for <code>IT_Vector<T></code>	418
Table 21: Artix Mapping of IDL Basic Types to C++	424
Table 22: Basic <code>IT_Bus::Var<T></code> Operations	446
Table 23: Non-Atomic Built-In Types Supported by Reflection	454
Table 24: Effect of nillable, minOccurs and maxOccurs Settings	473

LIST OF TABLES

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.

If you would like to know more about WSDL concepts, see the Introduction to WSDL in *Learning about Artix*.

Finding Your Way Around the Library

The Artix library contains several books that provide assistance for any of the tasks you are trying to perform. The Artix library is listed here, with a short description of each book.

If you are new to Artix

You may be interested in reading:

- [Release Notes](#) contains release-specific information about Artix.
- [Installation Guide](#) describes the prerequisites for installing Artix and the procedures for installing Artix on supported systems.
- [Getting Started with Artix](#) describes basic Artix and WSDL concepts.

To design and develop Artix solutions

Read one or more of the following:

- [Designing Artix Solutions](#) provides detailed information about describing services in Artix contracts and using Artix services to solve problems.
- [Developing Artix Applications in C++](#) discusses the technical aspects of programming applications using the C++ API.
- [Developing Artix Plug-ins with C++](#) discusses the technical aspects of implementing plug-ins to the Artix bus using the C++ API.
- [Developing Artix Applications in Java](#) discusses the technical aspects of programming applications using the Java API.
- [Artix for CORBA](#) provides detailed information on using Artix in a CORBA environment.
- [Artix for J2EE](#) provides detailed information on using Artix to integrate with J2EE applications.
- [Artix Technical Use Cases](#) provides a number of step-by-step examples of building common Artix solutions.

To configure and manage your Artix solution

Read one or more of the following:

- [Deploying and Managing Artix Solutions](#) describes how to deploy Artix-enabled systems, and provides detailed examples for a number of typical use cases.
- [Artix Configuration Guide](#) explains how to configure your Artix environment. It also provides reference information on Artix configuration variables.
- [IONA Tivoli Integration Guide](#) explains how to integrate Artix with IBM Tivoli.
- [IONA BMC Patrol Integration Guide](#) explains how to integrate Artix with BMC Patrol.
- [Artix Security Guide](#) provides detailed information about using the security features of Artix.

Reference material

In addition to the technical guides, the Artix library includes the following reference manuals:

- [Artix Command Line Reference](#)
- [Artix C++ API Reference](#)

- [Artix Java API Reference](#)

Have you got the latest version?

The latest updates to the Artix documentation can be found at <http://www.iona.com/support/docs>.

Compare the version dates on the web page for your product version with the date printed on the copyright page of the PDF edition of the book you are reading.

Searching the Artix Library

You can search the online documentation by using the **Search** box at the top right of the documentation home page:

<http://www.iona.com/support/docs>

To search a particular library version, browse to the required index page, and use the **Search** box at the top right. For example:

<http://www.iona.com/support/docs/artix/3.0/index.xml>

You can also search within a particular book. To search within an HTML version of a book, use the **Search** box at the top left of the page. To search within a PDF version of a book, in Adobe Acrobat, select **Edit|Find**, and enter your search text.

Online Help

Artix Designer includes comprehensive online help, providing:

- Detailed step-by-step instructions on how to perform important tasks.
- A description of each screen.
- A comprehensive index, and glossary.
- A full search feature.
- Context-sensitive help.

There are two ways that you can access the online help:

- Click the Help button on the Artix Designer panel, or
- Select **Contents** from the Help menu

Additional Resources

The [IONA Knowledge Base](http://www.iona.com/support/knowledge_base/index.xml) (http://www.iona.com/support/knowledge_base/index.xml) contains helpful articles written by IONA experts about Artix and other products.

The [IONA Update Center](http://www.iona.com/support/updates/index.xml) (<http://www.iona.com/support/updates/index.xml>) contains the latest releases and patches for IONA products.

If you need help with this or any other IONA product, go to [IONA Online Support](http://www.iona.com/support/index.xml) (<http://www.iona.com/support/index.xml>).

Comments, corrections, and suggestions on IONA documentation can be sent to docs-support@iona.com.

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

Fixed width Fixed width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `IT_Bus: AnyType` class.

Constant width paragraphs represent code examples or information a system displays on the screen. For example:

```
#include <stdio.h>
```

Fixed width italic Fixed width italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example:

```
% cd /users/YourUserName
```

Italic Italic words in normal text represent *emphasis* and introduce *new terms*.

Bold Bold words in normal text represent graphical user interface components such as menu commands and dialog boxes. For example: the **User Preferences** dialog.

Keying Conventions

This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, the command prompt is not shown.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the MS-DOS or Windows command prompt.
...	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
	In format and syntax descriptions, a vertical bar separates items in a list of choices enclosed in { } (braces). In graphical user interface descriptions, a vertical bar separates menu commands (for example, select File Open).

PREFACE

Developing Artix Enabled Clients and Servers

Artix generates stub and skeleton code that provides a developer with a simple model to develop transport independent applications.

In this chapter

This chapter discusses the following topics:

Generating Stub and Skeleton Code	page 2
C++ Namespaces	page 7
Defining a WSDL Interface	page 8
Developing a Server	page 10
Developing a Client	page 14
Generating a Sample Application from WSDL	page 19
Compiling and Linking an Artix Application	page 24
Building Artix Stub Libraries on Windows	page 26

Generating Stub and Skeleton Code

Overview

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:

<code>-i port_type</code>	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.
<code>-e web_service_name[:port_list]</code>	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.
<code>-b binding_name</code>	Specifies the name of the binding to use when generating code. The default is the first binding listed in the contract.
<code>-d output_dir</code>	Specifies the directory to which the generated code is written. The default is the current working directory.
<code>-n [URI=]C++namespace</code>	Maps an XML namespace to a C++ namespace. The C++ stub code generated from the XML namespace, <i>URI</i> , is put into the specified C++ namespace, <i>C++namespace</i> . This switch can appear multiple times.

<code>-nexclude</code> <code>URI [=C++namespace]</code>	Do not generate C++ stub code for the specified XML namespace, <code>URI</code> . You can optionally map the XML namespace, <code>URI</code> , to a C++ namespace, <code>C++namespace</code> , in case it is referenced by the rest of the XML schema/WSDL contract. This switch can appear multiple times.
<code>-ninclude</code> <code>URI [=C++namespace]</code>	Generates C++ stub code for the specified XML namespace, <code>URI</code> . You can optionally map the XML namespace, <code>URI</code> , to a C++ namespace, <code>C++namespace</code> . This switch can appear multiple times.
<code>-nimport</code> <code>C++namespace</code>	Specifies the C++ namespace to use for the code generated from imported schema.
<code>-impl</code>	Generates the skeleton code for implementing the server defined by the contract.
<code>-m {NMAKE UNIX}</code> <code>:[executable library]</code>	Used in combination with <code>-impl</code> to generate a makefile for the specified platform (<code>NMAKE</code> for Windows or <code>UNIX</code> for UNIX). You can specify that the generated makefile builds an executable, by appending <code>:executable</code> , or a library, by appending <code>:library</code> . For example, the options, <code>-impl -m NMAKE:executable</code> , would generate a Windows makefile to build an executable.
<code>-libv version</code>	Used in combination with either <code>-m NAME:library</code> or <code>-m UNIX:library</code> 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.
<code>-f</code>	<i>Deprecated</i> —No longer used (was needed to support routing in earlier versions).
<code>-server</code>	Generates stub code for a server (cannot be combined with the <code>-client</code> switch).
<code>-client</code>	Generates stub code for a client (cannot be combined with the <code>-server</code> switch).

<code>-sample</code>	<p>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.</p> <p>To generate a complete working sample application, combine <code>-sample</code> with the <code>-impl</code> and the <code>-m</code> switches.</p>
<code>-plugin</code> <code>[:plugin_name]</code>	<p>Generates servant registration code as a Bus plug-in. You can optionally specify the plug-in name by appending <code>:plugin_name</code> to this option. If no plug-in name is specified, the default name is <code><ServiceName><PortTypeName></code>. The service name, <code><ServiceName></code>, is specified by the <code>-e</code> option.</p> <p>See “Customizing servant registration” on page 21 for more details.</p>
<code>-deployable</code>	<p>(Used with <code>-plugin</code>.) Generates a deployment descriptor file, <code>deploy<ServiceName>.xml</code>, which is needed to deploy a plug-in into the Artix container.</p>
<code>-global</code>	<p>(Used with <code>-plugin</code>.) In the generated plug-in code, instantiate the plug-in using a <code>GlobalBusORBPlugIn</code> object instead of a <code>BusORBPlugIn</code> object.</p> <p>A <code>GlobalBusORBPlugIn</code> initializes the plug-in automatically, as soon as it is constructed (suitable approach for plug-ins that are linked directly with application code).</p> <p>A <code>BusORBPlugIn</code> is not initialized unless the plug-in is either listed in the <code>orb_plugins</code> list or deployed into an Artix container (suitable approach for dynamically loading plug-ins).</p>
<code>-v</code>	Displays the version of the tool.
<code>-license</code>	Displays the currently available licenses.
<code>-declspec declspec</code>	Creates Visual C++ declaration specifiers for <code>dlexport</code> and <code>dllimport</code> . This option makes it easier to package Artix stubs in a DLL library. See “Building Artix Stub Libraries on Windows” on page 26 for details.
<code>-all</code>	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 <code>wsdltocpp</code> is running.
-reflect	Enables reflection on generated data classes. See “Reflection” on page 441 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 39 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.

C++ Namespaces

Artix namespaces

Two built-in C++ namespaces widely used by the Artix runtime infrastructure are: `IT_Bus`, and `IT_WSDL`. The first namespace is used for the callable APIs and declarations, and the second is used for the functions that parse the WSDL at runtime; these are needed only by highly dynamic applications.

Solution specific namespaces

You can optionally instruct the C++ client proxy generator to put the proxy classes and complex data types into a custom C++ namespace. This is useful if you plan on using many Web services from a single client application. Consider the following sample application, where the `GroupB` service was put into a namespace called `GroupB`. Also note the use of the `IT_Bus` namespace for the data types.

```
#include "GroupBClient.h"
#include "GroupBClientTypes.h"

int main(int argc, char* argv[])
{
    GroupB::GroupBClient bc; // declare the client proxy class

    GroupB::SOAPStruct ssSend;
    ssSend.setvarFloat(IT_Bus::Float(5.67));
    ssSend.setvarInt(1234);
    ssSend.setvarString(IT_Bus::String("Embedded struct string"));

    IT_Bus::Int intValue = 0;
    IT_Bus::Float floatValue = IT_Bus::Float(0.0);

    IT_StringPtr pstring(bc.echoStructAsSimpleTypes(ssSend,
                                                    intValue, floatValue));
}
```

Defining a WSDL Interface

Overview

This section defines the `HelloWorld` port type, which is used as the basis for the server and client examples appearing in this chapter. The code for the `HelloWorld` demonstration is located in the following directory:

```
ArtixInstallDir/artix/Version/demos/basic/hello_world_soap_http
```

Restrictions

The following restrictions currently apply when defining a WSDL interface for Artix applications:

- Some simple atomic types are not supported—see [“Unsupported Simple Types”](#) on page 287.
-

WSDL example

[Example 1](#) shows the WSDL for a `HelloWorld` port type, which defines two operations, `greetMe` and `sayHi`.

Example 1: WSDL Definition of the `HelloWorld` Port Type

```
// C++
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloWorldService"
  targetNamespace="http://xmlbus.com/HelloWorld"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://xmlbus.com/HelloWorld"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
  <message name="greetMe">
    <part name="stringParam0" type="xsd:string"/>
  </message>
  <message name="greetMeResponse">
    <part name="return" type="xsd:string"/>
  </message>
  <message name="sayHi"/>
  <message name="sayHiResponse">
    <part name="return" type="xsd:string"/>
  </message>
  <portType name="HelloWorldPortType">
    <operation name="greetMe">
      <input message="tns:greetMe" name="greetMe"/>
      <output message="tns:greetMeResponse" name="greetMeResponse"/>
    </operation>
  </portType>
</definitions>
```


Example 1: *WSDL Definition of the HelloWorld Port Type*

```
        name="greetMeResponse" />
    </operation>
    <operation name="sayHi">
        <input message="tns:sayHi" name="sayHi" />
        <output message="tns:sayHiResponse"
            name="sayHiResponse" />
    </operation>
</portType>
<binding ... >
    ...
</binding>
<service name="HelloWorldService">
    ...
</service>
</definitions>
```

Developing a Server

Overview

The Artix code generator generates server skeleton code and the implementation shell that serves as the starting point for developing a server that uses the Artix Bus. This skeleton code hides the transport details from the application developer, allowing them to focus on business logic.

Generating the server implementation class

The Artix code generator utility, `wsdltocpp`, will generate an implementation class for your server when passed the `-impl` command flag.

Generated code

The implementation class code consists of two files:

`PortTypeNameImpl.h` contains the signatures and data types needed for the server implementation.

`PortTypeNameImpl.cxx` contains empty shells for the methods that implement the operations defined in the contract, as well as an empty constructor and destructor for the impl class. This file also contains a factory class for the server implementation.

Completing the server implementation

You must provide the logic for the operations specified in the contract that defines the server. To do this you edit the empty methods provided in `PortTypeNameImpl.cxx`. The generated impl class, `HelloWorldImpl.cxx`, for the contract defined in this chapter would resemble [Example 2](#). The majority of the code in [Example 2](#) is auto-generated by the WSDL-to-C++ compiler. Only the code portions highlighted in `bold` (in the bodies of the `greetMe()` and `sayHi()` functions) must be inserted by the programmer.

Example 2: Implementation of the HelloWorld Port Type in the Server

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

IT_USING_NAMESPACE_STD
using namespace IT_Bus;
```

Example 2: *Implementation of the HelloWorld Port Type in the Server*

```

HelloWorldImpl::HelloWorldImpl(IT_Bus::Bus_ptr bus,
    IT_Bus::Port* port)
    : HelloWorldServer(bus,port)
{
}

HelloWorldImpl::~HelloWorldImpl()
{
}

void
HelloWorldImpl::greetMe(
    const IT_Bus::String & stringParam0,
    IT_Bus::String & Response
) IT_THROW_DECL((IT_Bus::Exception))
{
    cout << "HelloWorldImpl::greetMe called with message: "
        << stringParam0 << endl;
    Response = IT_Bus::String("Hello Artix User: ")+stringParam0;
}

void
HelloWorldImpl::sayHi(
    IT_Bus::String & Response
) IT_THROW_DECL((IT_Bus::Exception))
{
    cout << "HelloWorldImpl::sayHi called" << endl;
    Response = IT_Bus::String("Greetings from the Artix
HelloWorld Server");
}

```

Writing the server main()

The server `main()` handles the initialization of the Artix Bus, the running of the Artix Bus, and the shutdown of the Artix Bus.

Initializing the Bus

The Bus is initialized using `IT_Bus::init()`. The method has the following signature:

```

static Bus& init(int argc,
                char* argv[],
                const char* scope = "");

```

The third parameter is optional and is used to identify the configuration scope used by the Bus for this application.

Example 3 shows an example of initializing the Artix bus in a server. It is important to retain an instance of the initialized Bus as it is needed to register your server implementation factories,

Example 3: *Initializing the Artix Bus in a Server main()*

```
// C++
IT::Bus_var bus = IT_Bus::init(argc, argv);
```

Registering the Servant Objects

To make the `HelloWorldImpl` servant object accessible to remote clients, you must register it with the Bus instance. Registration also has the side effect of activating the associated WSDL service, `service_name`.

Example 4: *Registering a Servant Object for HelloWorld*

```
// C++
// demos/servant_management/transient_servants/server/server.cxx
...
try {
    ...
    HelloWorldImpl servant(bus);

    QName service_name("", "HelloWorldService",
        "http://xmlbus.com/HelloWorld");

    bus->register_servant(
        servant,
        "./hello_world.wsdl",
        service_name,
        "HelloWorldPort"
    );
    ...
} catch (IT_Bus::Exception& e) { ... }
```

Running the Bus

After the Bus is initialized it is ready to listen for requests and pass them to the server for processing. To start the Bus, you use `IT_Bus::run()`. Once the Bus is started, it retains control of the process until it is shut down. The server's `main()` will be blocked until `run()` returns.

Shutting the Bus down

Because `IT_Bus::run()` never returns control to the server's `main()`, you must kill the server process (for example, using Ctrl-C) to shut down the server.

Completed server main()

[Example 5 on page 13](#) shows how the `main()` for the server defined by the HelloWorld contract might look.

Example 5: *ConverterServer main()*

```
// C++
#include <it_bus/bus.h>
#include <it_bus/Exception.h>
#include <it_bus/fault_exception.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);

        HelloWorldImpl servant(bus);

        QName service_name("", "HelloWorldService",
            "http://xmlbus.com/HelloWorld");

        bus->register_servant(
            servant,
            "./hello_world.wsdl",
            service_name,
            "HelloWorldPort"
        );

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

    return 0;
}
```

Developing a Client

Overview

The stub code for a client implementation for the service defined by the contract is contained in the files `PortTypeNameClient.h` and `PortTypeNameClient.cxx`. You should never make any modifications to the generated code in these files. You also need to reference the files `PortTypeName.h` and `PortTypeNameTypes.h` in your client code.

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

Initializing the Bus

Client applications initialize the bus in the same manner as server applications, by calling `IT_Bus::init()`. Client applications, however, do not need to make a call to `IT_Bus::run()`.

Instantiating the client object

The generated `HelloWorld` client proxy object has constructors as shown in [Example 6 on page 14](#).

Example 6: *Generated Client Constructors*

```
HelloWorldClient();

HelloWorldClient(const IT_Bus::String & wsdl);

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

HelloWorldClient(const IT_Bus::Reference & reference);
```

Constructor with no arguments

The first constructor for the client proxy class takes no parameters. When using this constructor, the client requires that the contract defining its behavior be located in the same directory as the executable. The client uses the port and service specified at code generation time using the `-t` and `-b` flags.

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 port and service specified at code generation time using the `-t` and `-b` flags. This is useful for situations where the contracts are stored in a central location.

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 98](#)).

Constructor with three arguments

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

<code>wSDL</code>	Specifies the URL of the contract defining the client's behavior.
<code>service_name</code>	Specifies the name of the service, defined in the contract with a <code><service></code> tag, to use when connecting to the server.
<code>port_name</code>	Specifies the name of the port, defined in the contract with a <code><port></code> tag, to use when connecting to the server. The port name given must be defined in the specified <code><service></code> tag.

The client code is binding and transport neutral. Hence, the only restriction in specifying the port to use is that it have the same `portType` as the generated proxy. The port details are read in from the WSDL contract file at runtime. For example, if the contract for the conversion service is modified to include a service definition like the one shown in [Example 7 on page 15](#), you could instantiate the client proxy to use either HTTP or Tuxedo.

Example 7: Multiple Ports Defined for HelloWorld

```
<service name="HelloWorldService2">
  <port name="HelloWorldHTTPPort"
    binding="tns:HelloWorldBinding">
    <soap:address location="http://localhost:8081"/>
  </port>
  <port name="HelloWorldTuxedoPort"
    binding="tns:HelloWorldBinding">
    <tuxedo:address serviceName="TuxQueue"/>
  </port>
</service>
```

To specify that the proxy client is to connect to the server using the Tuxedo server `TuxQueue`, you would instantiate the client using the following constructor:

```
HelloWorldClient proxy("HelloWorld.wsdl", "HelloWorldService2",
    "HelloWorldTuxedoPort");
```

Constructor with a reference argument

The fourth constructor takes one argument representing an Artix reference, `IT_Bus::Reference`. The Artix reference contains complete service and port details, including addressing information, enabling the client proxy to open a connection to a remote service. For a detailed discussion of Artix references, see [“Artix References” on page 103](#).

Invoking the operations

To invoke the operations offered by the service, the client calls the methods of the client proxy object. The generated client proxy class contains one method for each operation defined in the contract. The generated methods all return void. Any response messages are passed by reference as a parameter to the method. For example, the `greetMe` operation defined in [Example 1](#) generates a method with the following signature:

```
void greetMe(
    const IT_Bus::String & stringParam0,
    IT_Bus::String & var_return
) IT_THROW_DECL((IT_Bus::Exception));
```

Shutting the bus down

Unlike a server that must shut down the bus from a separate thread, clients do not typically make a call to `IT_Bus::run()` and can simply call `IT_Bus::shutdown()` before the main thread exits. It is advisable to pass `TRUE` to `IT_Bus::shutdown()` to ensure that the bus is fully shut down before exiting.

Full client code

A client developed to access the service defined by the `HelloWorldService` contract will look similar to [Example 8](#).

Example 8: *HelloWorld Client*

```
// C++
#include <it_bus/bus.h>
#include <it_bus/Exception.h>
```


Example 8: *HelloWorld Client*

```

#include <it_cal/iostream.h>
1 #include "HelloWorldClient.h"

IT_USING_NAMESPACE_STD
using namespace IT_Bus;

using namespace HW;

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

    try
    {
2         IT_Bus::init(argc, argv);
3         HelloWorldClient hw;

        String string_in;
        String string_out;

4         hw.sayHi(string_out);
        cout << "sayHi method returned: " << string_out << endl;

        if (argc > 1) {
            string_in = argv[1];
        } else {
            string_in = "Early Adopter";
        }
        hw.greetMe(string_in, string_out);
        cout << "greetMe method returned: " << string_out << endl;
    }
5 catch(IT_Bus::Exception& e)
    {
        cout << endl << "Caught Unexpected Exception: "
            << endl << e.Message()
            << endl;
        return -1;
    }

    return 0;
}

```

The code does the following:

1. The `PortNameClient.h` header includes the definitions for the client proxy class.
2. The `IT_Bus::init()` static function initializes the bus.
3. 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.
4. Invoke the `sayHi()` operation on the client proxy.
5. 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`.

Generating a Sample Application from WSDL

Overview

You can use the WSDL-to-C++ compiler to generate a working Web service application, consisting of a sample client application and a sample server application. You can then finish the application by editing the default client and server code. This approach enables you to develop a Web service application rapidly.

Sample WSDL file

The examples in this section are based on the `hello_world.wsdl` file, located in the following directory:

```
ArtixInstallDir/artix/Version/demos/basic/hello_world_soap_http/e
tc
```

Generating the sample application

To generate a complete sample application from the `hello_world.wsdl` file, including a client and a server, enter the following command:

Windows

```
> wsdltocpp -sample -impl -m NMAKE:executable -plugin -global
hello_world.wsdl
```

UNIX

```
% wsdltocpp -sample -impl -m UNIX:executable -plugin -global
hello_world.wsdl
```

Generated files

The preceding `wsdltocpp` command generates the following files:

Stub Files

```
<PortType>.h
<PortType>Client.h
<PortType>Server.h
<PortType>Client.cxx
<PortType>Server.cxx
<WSDLFileName>_wsdlTypesFactory.h
<WSDLFileName>_wsdlTypesFactory.cxx
```

Client Implementation Files

```
<PortType>ClientSample.cxx
```

Server Implementation Files

```
<PortType>Impl.h
<PortType>Impl.cxx
<PortType>ServerSample.cxx
<ServiceName><PortType>PlugIn.cxx
```

Makefile

```
Makefile
```

Building the sample application

With the help of the generated makefile, `Makefile`, you can build the client and server applications as follows:

Windows

```
> nmake all
```

UNIX

```
% make all
```

Customizing the servant implementation

To complete the server implementation, you should edit the `<PortType>Impl.h` file to fill in the missing operations in the `<PortType>Impl` servant class.

For example, [Example 9](#) shows the generated servant class, `GreeterImpl`, that implements the `Greeter` port type. To complete the sample implementation, you should insert code after the `// User code goes in here` comments (highlighted in bold font in [Example 9](#)).

Example 9: Generated Implementation of the Greeter Port Type

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

GreeterImpl::GreeterImpl(IT_Bus::Bus_ptr bus) :
    GreeterServer(bus)
{
}

GreeterImpl::~GreeterImpl()
{
}

IT_Bus::Servant*
```

Example 9: *Generated Implementation of the Greeter Port Type*

```
GreeterImpl::clone() const
{
    return new GreeterImpl(get_bus());
}

void
GreeterImpl::sayHi (
    IT_Bus::String &theResponse
) IT_THROW_DECL((IT_Bus::Exception))
{
    // User code goes in here
}

void
GreeterImpl::greetMe (
    const IT_Bus::String &me,
    IT_Bus::String &theResponse
) IT_THROW_DECL((IT_Bus::Exception))
{
    // User code goes in here
}
```

Customizing servant registration

To activate a particular Web service, you must register a servant instance with the Artix Bus. In a generated application, the servant registration code appears in the `<ServiceName><PortType>PlugIn.cxx` file, which embeds the servant registration code in an Artix plug-in.

For example, if you generate a sample application from `hello_world.wsdl` (passing the `-plugin` and `-global` flags to `wsdltocpp`), you obtain the file, `SOAPServiceGreeterPlugIn.cxx`, which defines the `GreeterServantBusPlugIn` plug-in class. [Example 10](#) is an extract from the `SOAPServiceGreeterPlugIn.cxx` file that shows the servant registration code.

Example 10: *Extract from the GreeterServantBusPlugIn Class*

```
// C++
...
GreeterServantBusPlugIn::GreeterServantBusPlugIn (
    Bus_ptr bus
) IT_THROW_DECL((Exception))
:
```

Example 10: *Extract from the GreeterServantBusPlugIn Class*

```

    BusPlugIn(bus),
    m_servant(bus),
    m_service_qname("", "SOAPService",
    "http://www.ionas.com/hello_world_soap_http")
{
    // complete
}

GreeterServantBusPlugIn::~GreeterServantBusPlugIn()
{
    // complete
}

void
GreeterServantBusPlugIn::bus_init(
) IT_THROW_DECL( (Exception) )
{
    WSDLService* wsdl_service =
        get_bus()->get_service_contract(m_service_qname);
    if (wsdl_service != 0)
    {
        get_bus()->register_servant(
            m_servant,
            *wsdl_service
        );
    }
    else
    {
        get_bus()->register_servant(
            m_servant,
            "hello_world.wsdl",
            m_service_qname
        );
    }
}
...

```

If you want to change the details of servant registration, you can edit the `register_servant()` calls in the `SOAPServiceGreeterPlugIn.cpp` file. For a detailed discussion of servant registration, see [“Registering Servants” on page 64](#).

Automatic plug-in activation

In order to have any effect, an Artix plug-in must register itself with the Artix Bus and the Bus must be configured to activate the plug-in. When you generate a plug-in class using the `-plugin` and `-global` flags, however, registration and activation of the plug-in occur automatically.

For example, the `SOAPServiceGreeterPlugIn.cxx` file includes the following call to construct a `GlobalBusORBPlugIn` object:

```
// C++
GlobalBusORBPlugIn bus_plugin_SOAPServiceGreeter (
    "SOAPServiceGreeter",
    plugin_factory_SOAPServiceGreeter
);
```

The `GlobalBusORBPlugIn` is an object that automatically registers and activates the plug-in (whose name is given by the string, `SOAPServiceGreeter`). In contrast to regular plug-in objects, of `BusORBPlugIn` type, it is *not* necessary to activate the plug-in by adding the plug-in name to the `orb_plugins` list; activation of `GlobalBusORBPlugIn` objects is automatic.

Compiling and Linking an Artix Application

Compiler Requirements

An application built using Artix requires a number of IONA-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 <code>IT_Bus::Float</code> , etc. Required for all applications that use Artix functionality.
it_ifc.lib	libit_ifc.so libit_ifc.sl libit_ifc.a	The IONA foundation classes provide IONA specific data types and exceptions.
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) \artix\$(IT_PRODUCT_VER) \bin"
```

and

```
"$(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

Overview

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
#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.

Artix Programming Considerations

Several areas must be considered when programming complex Artix applications.

In this chapter

This chapter discusses the following topics:

Bootstrapping Service	page 28
Operations and Parameters	page 34
Exceptions	page 44
Memory Management	page 51
Registering Servants	page 64
Multi-Threading	page 80
Converting with <code>to_string()</code> and <code>from_string()</code>	page 92
Locating Services with UDDI	page 98
Overriding a HTTP Address in a Client	page 100

Bootstrapping Service

Overview

When it comes to deploying applications in a real system, it is typically inconvenient to hardcode the location of a WSDL contract in the application. It is more practical to specify the location of basic resources, such as a WSDL contract, at runtime—for example, by specifying the WSDL contract URL in configuration or on the command line.

The Artix *bootstrapping service* simplifies the process of obtaining the following kinds of basic resource: WSDL contracts and Artix references. The process is divided into two independent steps:

1. *Provide the basic resource*—you can provide a WSDL contract or an Artix reference in several different ways: by configuration, by specifying the location on the command line, and so on.
2. *Retrieve the basic resource*—C++ functions are provided to retrieve WSDL services and Artix references, based on the qualified name (QName) of the resource.

In this section

This section contains the following subsections:

How Clients Find Initial References	page 29
How Servers Find WSDL Contracts	page 31

How Clients Find Initial References

Overview

An Artix reference encapsulates the data required for opening a connection to an Artix endpoint (essentially, this data is identical to the data contained in a WSDL `service` element). Once a client has a reference, it can easily open a connection to a remote service by passing the reference to a proxy constructor.

The Artix bootstrapping service provides an API function, `IT_Bus::resolve_initial_references()`, for finding initial references based on the QName of a WSDL service.

Example of finding an initial reference

Given that the Bus has already loaded and parsed either an Artix reference (or a WSDL contract) containing a service called `SOAPService` in the namespace, `http://www.iona.com/hello_world_soap_http`, you can initialize a client proxy, `proxy`, as follows:

Example 11: *Finding an Initial Reference Using the Bootstrapping Service*

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

// 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();
```

Options for bootstrapping references

The bootstrapping service 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 Artix reference. For example:

```
./server -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 Artix reference file or by specifying the literal value of an Artix reference.

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

5. *Service in a WSDL contract*—the `service` element in a WSDL contract contains essentially the same data as an Artix 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 bootstrapping references is, thus, effectively an extension of the mechanism for bootstrapping WSDL contracts—see [“Options for bootstrapping WSDL” on page 32](#).

How Servers Find WSDL Contracts

Overview

You need to locate the requisite WSDL contract before you can register a servant with the Bus. The effect of registration is to associate an implementation (represented by a servant object) with a particular WSDL service. The WSDL service must, therefore, be available from one of the WSDL contracts provided by the bootstrapping service.

The Artix bootstrapping service provides an API function, `IT_Bus::get_service_contract()`, for retrieving WSDL service elements from a WSDL contract.

Example of finding a WSDL contract

Given that the Bus has already loaded and parsed a WSDL contract containing the service, `SOAPService`, in the namespace, `http://www.ionas.com/hello_world_soap_http`, you can find the WSDL service element and register a servant against it as follows:

Example 12: *Finding a WSDL Contract Using the Bootstrapping Service*

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

// Find the WSDL contract using the bootstrap service
IT_WSDL::WSDLService* wsd_service = bus->get_service_contract(
    service_qname
);

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

Options for bootstrapping WSDL

The bootstrapping service 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:

```
./server -BUSinitial_contract ../../etc/hello.wsdl
```
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}SOAPService";
bus:initial_contract:url:hello_service =
  "../../etc/hello.wsdl";
```

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

For more details, see *Deploying and Managing 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 the bootstrapping service looks for a particular WSDL service, it searches all of the WSDL files in the specified directory. For example:

```
./server -BUSservice_contract_dir ../../etc/
```

For more details, see *Deploying and Managing Artix Solutions*.
4. *Contract directory specified in the configuration file*—you can provide a WSDL contract by specifying a contract directory in the configuration file. For example:

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


5. *Stub WSDL shared library*—the bootstrapping service 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 Servants” on page 64](#).

Operations and Parameters

Overview

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.

In this section

This section contains the following subsections:

RPC/Literal Style	page 35
Document/Literal Wrapped Style	page 39

RPC/Literal Style

Overview

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 13 on page 36](#)).
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 13 on page 36](#)).
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 13](#) shows an example of a simple operation, `testParams`, which takes two input parameters, `inInt` and `inoutInt`, and two output parameters, `inoutInt` and `outFloat`.

Example 13: 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 14](#) shows how the preceding WSDL `testParams` operation (from [Example 13 on page 36](#)) maps to C++.

Example 14: 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 15](#) shows an example of an operation, `testReverseParams`, whose parameters are listed in the opposite order to that of the preceding `testParams` operation.

Example 15: 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 16](#) shows how the preceding WSDL `testReverseParams` operation (from [Example 15](#) on page 37) maps to C++.

Example 16: 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 elements 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

Overview

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, *OperationNameResult*.
- 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 17 on page 41](#)).
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 17 on page 41](#)).
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 17 on page 41](#)).

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 17 on page 41](#)).
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 13](#) shows an example of a simple operation, `testParams`, which takes two input parameters, `inInt` and `inoutInt`, and two output parameters, `inoutInt` and `outFloat`.

Example 17: `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>
  </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">
```

Example 17: *testParams Operation in Document/Literal Wrapped Style*

```

        <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 18](#) shows how the preceding WSDL `testParams` operation (from [Example 17 on page 41](#)) maps to C++.

Example 18: *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 17 on page 41](#) 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 19](#) shows the result of mapping the WSDL `testParams` operation to C++ with the `-wrapped` flag:

Example 19: C++ Mapping Using the *-wrapped* Flag

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

Exceptions

Overview

Artix provides a variety of built-in exceptions, which can alert users to problems with network connectivity, parameter marshalling, 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.

In this section

This section contains the following subsections:

Built-In Exceptions	page 45
User-Defined Exceptions	page 47

Built-In Exceptions

Overview

The Artix libraries and generated code generate exceptions from classes based on `IT_Bus::Exception`, defined in `<it_bus/Exception.h>`. `IT_Bus::Exception` provides all Artix built-in exceptions with the following methods for providing information back to the user:

`IT_Bus::Exception::message()`

`message()` returns an informative description of the error which generated the exception. It has the following signature:

```
const char* message() const;
```

Exception types

Artix defines the following exception types:

`IT_Bus::ServiceException` is thrown when there is a problem creating a Service. It is defined in `<it_bus/service_exception.h>`.

`IT_Bus::IOException` is thrown if there is an error writing a wsdl model to a stream. It is defined in `<it_bus/io_exception.h>`.

`IT_Bus::TransportException` is thrown if there is a communication failure. It is defined in `<it_bus/transport_exception.h>`.

`IT_Bus::ConnectException` is thrown if there is a communication error. This exception type is a specialization of a `TransportException`. It is defined in `<it_bus/connect_exception.h>`.

`IT_Bus::DeserializationException` is thrown if there is a problem unmarshaling data. Deserialization exceptions are propagated back to client stub code. It is defined in `<it_bus/deserialization_exception.h>`.

`IT_Bus::SerializationException` is thrown if there is a problem marshaling data. On the server-side if this is thrown as part of a dispatching an invocation the runtime will catch this and propagate a `Fault` to the client-side. On the client side these will get back to the application code. It is defined in `<it_bus/serialization_exception.h>`.

IT_Routing::InvalidRouteException is thrown if a route is improperly defined. It is defined in `<it_bus/invalid_route_exception.h>`.

User-Defined Exceptions

Overview

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::FaultException` class, which is defined in `<it_bus/fault_exception.h>`. The `FaultException` class extends `Exception`.

Declaring a fault in WSDL

[Example 20](#) 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 20: Declaration of the SampleFault Fault

```

1 <?xml version="1.0" encoding="UTF-8"?>
  <definitions ...>
    <types>
      <schema targetNamespace="http://soapinterop.org/xsd"
2         xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
        <complexType name="SampleFaultData">
          <all>
            <element name="lowerBound" type="xsd:int"/>
            <element name="upperBound" type="xsd:int"/>
          </all>
        </complexType>
        ...
      </schema>
    </types>
    <message name="SampleFault">
      <part name="exceptionData"
          type="xsd1:SampleFaultData"/>
    </message>
    ...
    <portType name="BasePortType">
      <operation name="echoInteger">
        <input message="tns:echoInteger" name="echoInteger"/>
        <output message="tns:echoIntegerResponse"

```

Example 20: Declaration of the *SampleFault* Fault

```

3      name="echoIntegerResponse"/>
      <fault message="tns:SampleFault"
            name="SampleFault"/>
    </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, `SampleFaultData` holds a lower bound and an upper bound).
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` element 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` elements that can be included in an `operation` element.

Raising a fault exception in a server

[Example 21](#) shows how to raise the `SampleFault` fault in the server code. The implementation of `echoInteger` now checks the input integer to see if it exceeds the given bounds.

The WSDL maps to C++ as follows:

- The WSDL `SampleFaultData` type maps to a C++ `SampleFaultData` class.
- The WSDL `SampleFault` message maps to a C++ `SampleFaultException` class. This follows the general pattern that `ExceptionMessage` maps to `ExceptionMessageException`.

Example 21: Raising the *SampleFault* Fault in the Server

```

// C++
void BaseImpl::echoInteger(const IT_Bus::Int
    inputInteger, IT_Bus::Int& Response)
    IT_THROW_DECL((IT_Bus::Exception))

```


Example 21: *Raising the SampleFault Fault in the Server*

```

{
    if (inputInteger<0 || 100<inputInteger)
    {
        // Create and initialize the SampleFaultData
        SampleFaultData ex_data;
        ex_data.setlowerBound(0);
        ex_data.setupperBound(100);

        // Create and initialize the fault.
        SampleFaultException ex;
        ex.setexceptionData(ex_data);

        // Throw the fault exception back to the client.
        throw ex;
    }
    cout << "BaseImpl::echoInteger called" << endl;
    Response = inputInteger;
}

```

Catching a fault exception in a client

[Example 22](#) shows how to catch the `SampleFault` fault on the client side. The client uses the proxy instance, `bc`, to call the `echoInteger` operation remotely.

Example 22: *Catching the SampleFault Fault in the Client*

```

// C++
...
try {
    Int int_out = 0;
    bc.echoInteger(int_in,int_out);
    if (int_in != int_out)
    {
        cout << endl << "echoInteger PASSED" << endl;
    }
}
catch (SampleFaultException &ex)
{
    cout << "Bounds exceeded:" << endl;
    cout << "lower bound = "
        << ex.getexceptionData().getlowerBound() << endl;
    cout << "upper bound = "
        << ex.getexceptionData().getupperBound() << endl;
}

```

Example 22: *Catching the SampleFault Fault in the Client*

```
catch (IT_Bus::FaultException &ex)
{
    /* Handle other fault exceptions ... */
}
catch (...)
{
    /* Handle all other exceptions ... */
}
```

Memory Management

Overview

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

In this section

This section contains the following subsections:

Managing Parameters	page 52
Assignment and Copying	page 57
Deallocating	page 59
Smart Pointers	page 60

Managing Parameters

Overview

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 60](#) 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 23](#) shows an example of a WSDL operation, `testSeqParams`, with three parameters, `inSeq`, `inoutSeq`, and `outSeq`, of sequence type, `xsd:SequenceType`.

Example 23: 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>
```

Example 23: WSDL Example with *in*, *inout* and *out* Parameters

```

</types>
...
<message name="testSeqParams">
  <part name="inSeq" type="xsd:SequenceType"/>
  <part name="inoutSeq" type="xsd:SequenceType"/>
</message>
<message name="testSeqParamsResponse">
  <part name="inoutSeq" type="xsd:SequenceType"/>
  <part name="outSeq" type="xsd: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 24](#) shows how to allocate, initialize, and deallocate parameters when calling the `testSeqParams` operation.

Example 24: 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);

```

Example 24: *Client Calling the testSeqParams Operation*

```

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 25](#) shows how the parameters are used on the server side, in the C++ implementation of the `testSeqParams` operation.

Example 25: *Server Calling the testSeqParams Operation*

```

// C++
void
BaseImpl::testSeqParams(
    const SequenceType & inSeq,
    SequenceType & inoutSeq,
    SequenceType & outSeq
) IT_THROW_DECL(IT_Bus::Exception)

```

Example 25: Server Calling the testSeqParams Operation

```

{
    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 57](#) 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

Overview

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 23 on page 52](#) 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 60](#).

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

Overview

To help you avoid memory leaks when using pointers, the WSDL-to-C++ compiler generates a smart pointer class, *ComplexTypePtr*, 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 26 shows how to call the `testSeqParams` operation using parameters that are allocated on the heap and referenced by *simple pointers*

Example 26: 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: "
         << 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 27](#) shows how to call the `testSeqParams` operation using parameters that are allocated on the heap and referenced by *smart pointers*

Example 27: 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.

Registering Servants

Overview

In order to make a servant accessible to remote clients, you must *register* the servant with a Bus instance. The effect of registration is twofold:

- A service is activated and begins listening for incoming requests.
- A servant object is linked to the newly-activated service. Requests received by the service are then dispatched to the linked servant object.

This section describes how to register servant objects with the `IT_Bus::Bus`; in particular, describing how to register both static and transient servants.

In this section

This section contains the following subsections:

Registering a Static Servant	page 65
Registering a Transient Servant	page 72

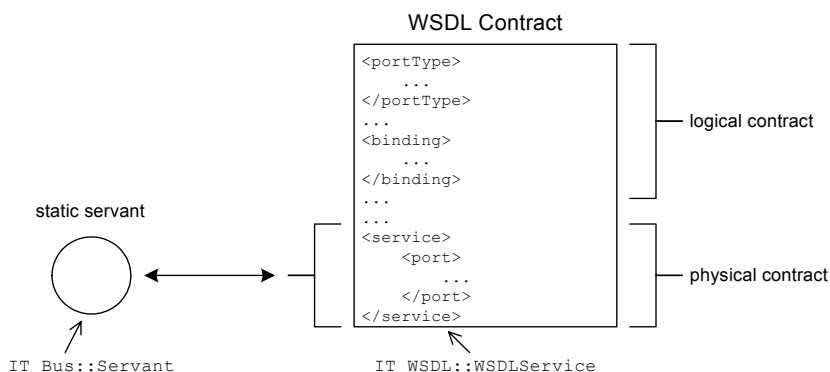
Registering a Static Servant

Overview

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 1 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 1: 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 28](#) to manage the registration of static servants:

Example 28: The `IT_Bus::Bus` Static Servant Registration API

```
// C++
void
register_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;

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 29: 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 several different approaches to activating a static 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 bootstrapping service. The following approaches are supported:

- [Activate all ports together and specify the WSDL location.](#)
- [Activate all ports together and use the bootstrapping service.](#)
- [Activate ports individually and specify the WSDL location.](#)
- [Activate ports individually and use the bootstrapping service.](#)

Activate all ports together and specify the WSDL location

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

bus->register_servant(
    bank_servant,
    "bank.wsdl",
    service_name
);
```

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

Note: If you need to obtain a reference to the resulting `IT_Bus::Service` instance, call `bus->get_service(service_name)`.

Activate all ports together and use the bootstrapping service

To use the bootstrapping service to activate a static servant's ports, call the `IT_Bus::get_service_contract()` function to obtain a pointer to a pre-existing WSDL service object. For example:

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

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

bus->register_servant(
    bank_servant,
    *wsdl_service
);
```

In this case, the WSDL contract containing the required WSDL service must already be loaded into the Artix Bus. The bootstrapping service provides several mechanisms for specifying the location of WSDL contracts. For more details, see [“How Servers Find WSDL Contracts” on page 31](#).

Activate ports individually and specify the WSDL location

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++
IT_Bus::QName service_name("", "BankService",
    "http://www.ionas.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");
```

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.

Activate ports individually and use the bootstrapping service

To use the bootstrapping service to activate a static servant's ports, call the `IT_Bus::get_service_contract()` function to obtain a pointer to a pre-existing WSDL service object. Registration is a two-step process. First you add a service to the Bus, then you activate individual ports. For example:

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

IT_WSDL::WSDLService* wsd_service = bus->get_service_contract(
    service_name
);

IT_Bus::Service_var bank_service =
    bus->add_service(*wsdl_service);
bank_service->register_servant(corba_servant, "CORBAPort");
bank_service->register_servant(soap_servant, "SOAPPort");
```

In this case, the WSDL contract containing the required WSDL service must already be loaded into the Artix Bus. The bootstrapping service provides several mechanisms for specifying the location of WSDL contracts. For more details, see [“How Servers Find WSDL Contracts” on page 31](#).

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 83](#) for more information.

Static servant example

Example 30 shows an example (taken from `demos/servant_management/transient_servants`) which shows how to register a servant as a static servant.

Example 30: Registering a Static Servant

```
// C++
// demos/servant_management/transient_servants/server/server.cxx
...
try {
    IT_Bus::Bus_var bus = IT_Bus::init(argc, (char **)argv);
1   BankImpl my_bank(bus);
2   QName service_name("", "BankService",
    "http://www.ionas.com/bus/demos/bank");
3   bus->register_servant(
        my_bank,
        "../wsdl/bank.wsdl",
        service_name
    );
4   IT_Bus::run();
5   bus->remove_service(service_name);
}
catch (IT_Bus::Exception& e) { ... }
```

The preceding code example can be explained as follows:

1. This line creates a servant instance, `my_bank`. At this point, we know that the servant implements the `Bank` port type (logical contract), but there is no association with any WSDL service (physical contract) yet.
2. This `IT_Bus::QName` instance refers to the `BankService` service from the WSDL contract. This is the WSDL service that will be associated with the servant.
3. The `register_servant()` function registers a static servant instance, taking the following arguments:
 - ◆ Servant instance.
 - ◆ WSDL file location.
 - ◆ Service QName.

The return value is an `IT_Bus::Service` object, which references the `BankService` WSDL service.

Immediately after registration, the service starts to process incoming invocations in a background thread.

4. The `IT_Bus::run()` function blocks the main thread of execution, allowing the registered services to continue processing incoming invocations in background threads.
5. The `remove_service()` function is called here to tidy up resources before the server shuts down. It deactivates the service and joins the background threads.

Registering a Transient Servant

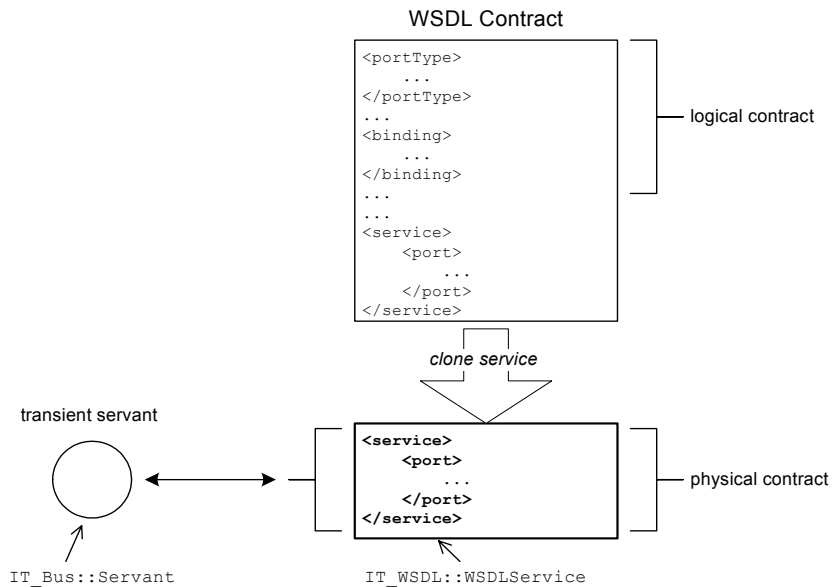
Overview

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 in the WSDL contract. This type of 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. If each customer's bank account maps to a service, it is clear that you require an unlimited number of services to represent customer accounts.

Figure 2 illustrates the effect of registering a transient servant: registration establishes an association between a servant instance and a cloned WSDL service.

Figure 2: Relationship between a Transient Servant and a WSDL Contract



Supported protocols

Artix currently supports transient servants for the following transports:

- HTTP.
 - CORBA.
 - Tunnel.
-

Transient servant

When a transient servant is registered, the following steps are implicitly performed by the `IT_Bus::Bus` instance (see [Figure 2](#)):

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.
-

Reuse of IP ports

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

IT_Bus::Bus transient registration functions

The `IT_Bus::Bus` class defines the functions in [Example 31](#) to manage the registration of transient servants.

Example 31: 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(
    IT_Bus::Servant & servant,
```

Example 31: *The IT_Bus::Bus Transient Servant Registration API*

```

    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 32: *The IT_Bus::Service register_servant() Function*

```

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

```

Example 32: *The `IT_Bus::Service register_servant()` Function*

```
);
```

Activating a static servant

There are several different approaches to activating a static 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 bootstrapping service. The following approaches are supported:

- [Activate all ports together and specify the WSDL location.](#)
 - [Activate all ports together and use the bootstrapping service.](#)
 - [Activate ports individually and specify the WSDL location.](#)
 - [Activate ports individually and use the bootstrapping service.](#)
-

Activate all ports together and specify the WSDL location

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:

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

IT_Bus::Service_var service =
    bus->register_transient_servant(
        account_servant,
        "bank.wsdl",
        service_name
    );
```

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

Activate all ports together and use the bootstrapping service

To use the bootstrapping service to activate a transient servant's ports, call the `IT_Bus::get_service_contract()` function to obtain a pointer to a pre-existing WSDL service object. For example:

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

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

IT_Bus::Service_var service =
    bus->register_transient_servant(
        account_servant,
        *wsdl_service
    );
```

In this case, the WSDL contract containing the required WSDL service must already be loaded into the Artix Bus. The bootstrapping service provides several mechanisms for specifying the location of WSDL contracts. For more details, see [“How Servers Find WSDL Contracts” on page 31](#).

Activate ports individually and specify the WSDL location

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:

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

IT_Bus::Service_var acc_service =
    bus->add_transient_service("bank.wsdl", service_name);
acc_service->register_servant(corba_servant, "CORBAPort");
acc_service->register_servant(soap_servant, "SOAPPORt");
```

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.

Activate ports individually and use the bootstrapping service

To use the bootstrapping service to activate a transient servant's ports, call the `IT_Bus::get_service_contract()` function to obtain a pointer to a pre-existing WSDL service object. Registration is a two-step process. First you add a service to the Bus, then you activate individual ports. For example:

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

IT_WSDL::WSDLService* wsd_service = bus->get_service_contract(
    service_name
);

IT_Bus::Service_var acc_service =
    bus->add_transient_service(*wsdl_service);
acc_service->register_servant(corba_servant, "CORBAPort");
acc_service->register_servant(soap_servant, "SOAPPort");
```

In this case, the WSDL contract containing the required WSDL service must already be loaded into the Artix Bus. The bootstrapping service provides several mechanisms for specifying the location of WSDL contracts. For more details, see [“How Servers Find WSDL Contracts” on page 31](#).

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 83](#) for more information.

Transient servant example

[Example 33](#) shows a sample implementation of the `Bank` port type's `create_account` operation (taken from `demos/servant_management/transient_servants`) which shows how to register a servant as a transient servant.

Example 33: Registering a Transient Servant

```
// C++
...
```

Example 33: *Registering a Transient Servant*

```

1  const IT_Bus::QName AccountImpl::SERVICE_NAME("",
    "AccountService", "http://www.iona.com/bus/demos/bank");
    ...
    void
    BankImpl::create_account(
        const IT_Bus::String &account_name,
        IT_Bus::Reference &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;

2          AccountImpl * new_account = new AccountImpl(
            get_bus(), account_name, 0
3          );
            Service_var service =
                get_bus()->register_transient_servant(
                    *new_account,
                    "../wsdl/bank.wsdl",
                    AccountImpl::SERVICE_NAME
                );

            // Now put the details for the account into the map so
            // we can retrieve it later.
            //
            AccountDetails details;
            details.m_service = service.release();
            details.m_account = new_account;

            account_iter = m_account_map.insert(
                AccountMap::value_type(account_name, details)
            ).first;
        }

        account_reference =
            (*account_iter).second.m_service->get_reference()
    }

```

The preceding C++ code can be described as follows:

1. The `AccountImpl::SERVICE_NAME` constant holds the qualified name of the `AccountService` service from the bank WSDL contract. This is the WSDL service that will be associated with the servant.
2. This line creates an `AccountImpl` servant instance, which implements the `Account` port type.
3. The `register_transient_servant()` function registers a transient servant instance, taking the following arguments:
 - ◆ Servant instance.
 - ◆ WSDL file location.
 - ◆ Service QName.

The return value is an `IT_Bus::Service` object, which references a WSDL service cloned from `AccountService`.

Multi-Threading

Overview

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

In this section

This section contains the following subsections:

Client Threading Issues	page 81
Servant Threading Models	page 83
Setting the Servant Threading Model	page 86
Thread Pool Configuration	page 89

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.

On the other hand, the client stub code is not inherently thread-safe. A single client proxy instance should not be shared amongst multiple threads without serializing access to the instance.

Single client proxy in two threads

[Example 34](#) below is a correctly written example featuring a single client proxy instance called from two different threads (assume `T1func` and `T2func` are called from two different threads):

Example 34: *Single Client Proxy in Two Threads*

```
#include <it_ts/mutex.h>
#include <it_ts/locker.h>

#include "BaseClient.h"
#include "BaseClientTypes.h"

BaseClient g_bc;
IT_Mutex mutexBC;

T1func ()
{
    IT_Locker<IT_Mutex> lock(mutexBC);
    g_bc.echoVoid();
}

T2func ()
{
    IT_Locker<IT_Mutex> lock(mutexBC);
    g_bc.echoVoid();
}
```

Two client proxies in two threads

[Example 35](#) below is another correctly written sample featuring two client proxy instances called from two different threads (assume `T1func` and `T2func` are called from two different threads):

Example 35: *Two Client Proxies in Two Threads*

```
#include "BaseClient.h"
#include "BaseClientTypes.h"
//nested inside BaseClient.h, may be omitted

T1func()
{
    BaseClient bc;
    bc.echoVoid();
}

T2func()
{
    BaseClient bc;
    bc.echoVoid();
}
```

Servant Threading Models

Overview

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 86). 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

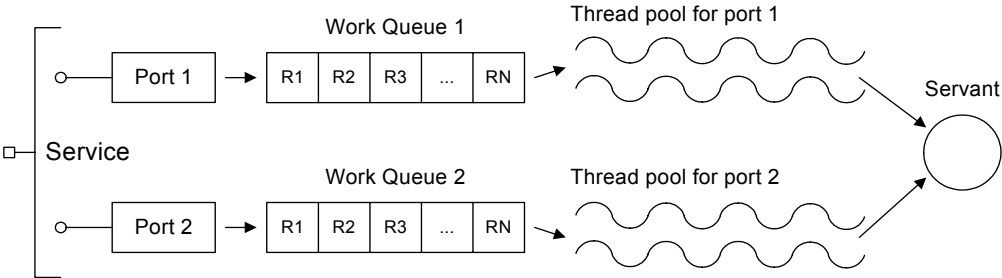
The default threading model is multi-threaded.

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 3](#) shows an outline of the multi-threaded threading model. In this case, the threads all share the same servant instance.

Figure 3: *Outline of the Multi-Threaded Threading Model*

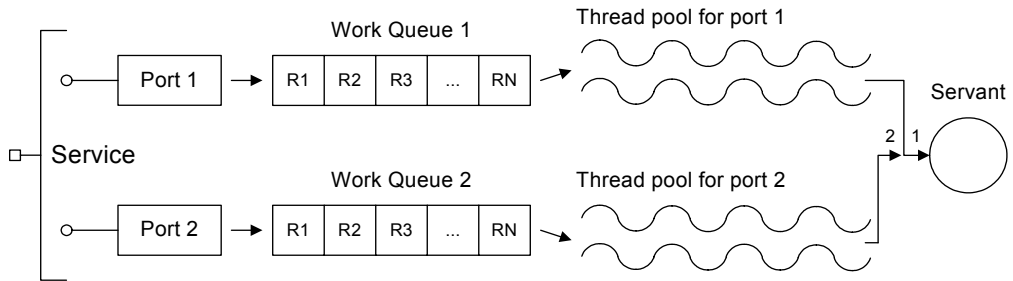


Serialized

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

Figure 4: *Outline of the Serialized Threading Model*

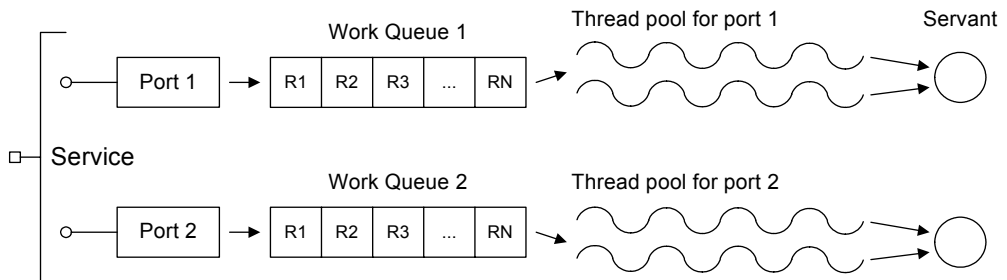


Per-port

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

Figure 5: *Outline of the Per-Port Threading Model*

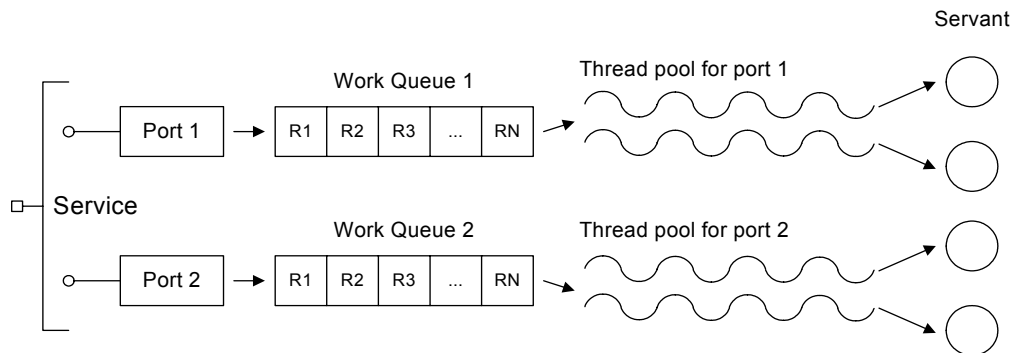


PerThread

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 6](#) 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 6: *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 6 on page 85](#)). 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

Overview

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 67](#) or [“Activating a static servant” on page 75](#)). 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.ionas.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 36](#) shows the constructors for each of the wrapper servant classes.

Example 36: 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 37](#) shows the default implementation of the `clone()` function for the servant class, `PortTypeImpl`.

Example 37: 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 38](#) shows how the main function of the bank server example can be modified to register the `BankImpl` servant with a `PerThread` threading model.

Example 38: 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 38: *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/artix/Version/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 `-ORBname 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 39](#) can be used to configure thread pools at the global level; that is, these settings would apply to all services by default.

Example 39: 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 40](#) show how to configure the thread pool for a service named `SessionManager`.

Example 40: 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 41](#) show how to configure the thread pool for a service named `SessionManager` in the `http://my.tns1/` namespace URI.

Example 41: 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()`

Overview

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/artix/Version/demos/basic/to_string`

Example struct

[Example 42](#) shows the definition of an XML schema type, `SimpleStruct`, which is used by the string conversion examples in this section.

Example 42: 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>
  </complexType>
</schema>
```

Example 42: *Schema Definition of a SimpleStruct Type*

```

        </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>3.1400
00105e0</varFloat><varInt>121</varInt><varString>goodbye</var
String></to_string>

```

In the stringified output, the element name defaults to `<to_string>`.

to_string()

Example 43 shows the signature of the `IT_Bus::to_string()` function, as defined in the `<it_bus/to_string.h>` header.

Example 43: 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>1.1000
00024e0</varFloat><varInt>2</varInt><varString>hello</varStri
ng></m1:foo>
```

In the stringified output, the element name is defined as `m1:foo`.

`from_string()`

[Example 44](#) shows the signature of the `IT_Bus::from_string()` function, as defined in the `<it_bus/from_string.h>` header.

Example 44: Signature of the `IT_Bus::from_string()` Function

```
// C++
namespace IT_Bus
{
    void IT_BUS_XML_API
```

Example 44: Signature of the `IT_Bus::from_string()` Function

```

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>1.10
000024e0</varFloat><varInt>2</varInt><varString>hello</varSt
ring></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

Overview

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) (<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=helloworld
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/artix/Version/demos/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"];
    ...
};

```

Overriding a HTTP Address in a Client

Overview

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 45](#) 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 45: 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 179](#)). 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 46 shows how to override the HTTP address to contact a `SOAPService` service running on the host, `yourhost`, and IP port, `5432`.

Example 46: 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>

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 =
```

Example 46: *Using HTTP_ENDPOINT_URL to Override a HTTP Address*

```
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 179](#).

Artix References

An Artix reference represents an endpoint. Because references can be passed around as parameters, they provide a convenient and flexible way of identifying and locating specific services.

In this chapter

This chapter discusses the following topics:

Introduction to References	page 104
The WSDL Publish Plug-In	page 106
References to Transient Services	page 113
Programming with References	page 116

Introduction to References

Overview

An Artix reference is an object that encapsulates endpoint and contract information for a particular WSDL service. References have the following features:

- A reference represents a `<wsdl:service>`.
- A reference is a built-in type in Artix.
- References can be sent across the wire as parameters of or return values from operations.
- References are fully self-describing. They contain endpoint and contract information in an optimized manner.
- References are the building blocks for the Artix *Locator* and the *Session Manager* services, because they enable you to create directories of Web services.
- References in Artix are protocol and transport neutral.

Note: The on-the-wire format of Artix 3.x references differs from the on-the-wire format of Artix 2.x references. By default, references generated by Artix 3.x applications cannot be parsed by Artix 2.x applications. If you need to enable interoperability between Artix 3.x and Artix 2.x, however, you can force Artix 3.x applications to generate the old on-the-wire format by including the following line in your Artix configuration file:

```
bus:reference_2.1_compat = "true";
```

Note: The Artix 2.x (and later) reference definition differs from the Artix 1.x reference definition. In Artix 1.x a reference is associated with a *WSDL port*, whereas in Artix 2.x a reference is associated with a *WSDL service* (which could contain multiple ports). Artix 2.x references are in line with the way WSDL 2.0 will handle service references.

Note: You cannot use references with rpc-encoded bindings, because references contain attributes, which are not compatible with rpc-encoding.

Contents of an Artix reference

An Artix reference encapsulates the following data:

- *Service QName*—the qualified name of the service with which the reference is associated.
- *WSDL location URL*—the server's copy of the WSDL contract. In a reference, the WSDL location URL serves two distinct purposes:
 - ◆ Service identification—the service is uniquely identified by the combination of a WSDL location URL and a service QName.
 - ◆ WSDL backup—allows the reference to be fully self-describing.

Note: If you have loaded the `wSDL_publish` plug-in on the server side, the WSDL location URL will point at a dynamically updated copy of the server's WSDL contract. See page 106.

- *List of ports*—an unbounded sequence of port elements, each of which contains the following data:
 - ◆ *Port name*—identifying the WSDL port.
 - ◆ *Binding QName*—the qualified name of the binding with which the port is associated.
 - ◆ *Properties*—a list of opaque properties, which makes the port element arbitrarily extensible. The properties list is typically used to hold transport-specific data and qualities of service. For example, if the port uses a SOAP binding, the properties would include a `soap:address` element specifying a host and IP port.

XML representation of a reference

The XML representation of a reference is defined by the following schema:

`ArtixInstallDir/artix/Version/schemas/references.xsd`

The schema is also available online at:

<http://schemas.iona.com/references/references.xsd>

The XML representation is used when marshaling or unmarshaling a reference as a WSDL parameter.

C++ representation of a reference

In C++, an Artix reference is represented by an instance of the `IT_Bus::Reference` class.

The WSDL Publish Plug-In

Overview

It is strongly recommended that you activate the *WSDL publish plug-in* for any applications that generate and export Artix 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 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 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

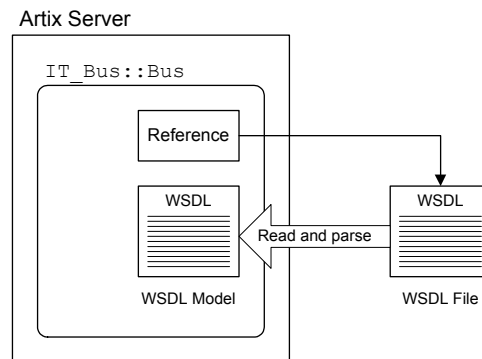
To load the WSDL publish plug-in, edit the `artix.cfg` configuration file and add `wSDL_publish` to the `orb_plugins` list in your application's configuration scope. For example, if your application's configuration scope is `demos.server`, you might use the following `orb_plugins` list:

```
# 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 7 gives an overview of how an Artix reference is generated when the WSDL publish plug-in is *not* loaded.

Figure 7: *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

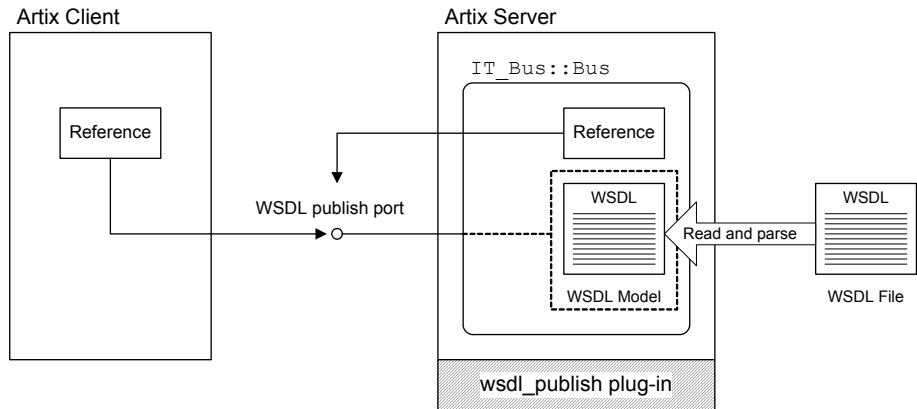
When an Artix server starts up, it reads the WSDL files needed by the registered services—for example, in Figure 7, a single WSDL file is read and parsed. After parsing, the WSDL definitions exist in memory in the form of a *WSDL model*. The WSDL model is an XML parse tree containing all the WSDL definitions 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 8](#) gives an overview of how an Artix reference is generated when the WSDL publish plug-in is loaded.

Figure 8: *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://hostname:WSDLPublishPort/QueryString
```

Where *hostname* is the server host, *WSDLPublishPort* 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 plug-in” on page 109](#)).

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 plug-in

Assume you configured `wSDL_publish` using the following values on a system with the IP address 10.1.2.3:

```
scope
{
  plugins:wSDL_publish:publish_port = 1234;
  plugins:wSDL_publish:hostname = "ipaddress";
};
```

The `wSDL_publish` base URL will be `http://10.1.2.3:1234`. Requests on the following types of URLs will be serviced:

- `http://10.1.2.3:1234/get_wSDL`, `http://10.1.2.3:1234/get_wSDL/`, `http://10.1.2.3:1234/get_wSDL?`, or `http://10.1.2.3:1234/get_wSDL/?` will return the HTML Menu. See [“Using the WSDL publish HTML menu” on page 109](#).
- `http://10.1.2.3:1234/get_wSDL?service=name&scope=EncodedUrl` will return the contract for the service specified in the query string.
- `http://10.1.2.3:1234/get_wSDL?stub=EncodedUrl` will return the contract for IONA specific services.
- `http://10.1.2.3:1234/inspection.wsil` will return a WSIL document containing information about active web services. See [“WSIL support” on page 110](#).
- `http://10.1.2.3:1234/get_wSDL/context/filename.wSDL` will return the specified wSDL contract. The value of `context` is generated at runtime.
- `http://10.1.2.3:2000/service` or `http://10.1.2.3:2000/service?wSDL` will return the contract for the specified service. The value of the URL is the same as the one specified in the WSDL as the `soap:address` of the service.

If an invalid URL is provided, `wSDL_publish` will return an HTTP 404 (Not Found) Error.

Using the WSDL publish HTML menu

The HTML menu provided by `wSDL_publish` is an HTML page that contains links to the contracts of all services activated on the current bus associated with the specified `wSDL_publish` instance.

Assuming an `it_container` instance is started on port 2000. The HTML menu available at `http://10.1.2.3:1234/get_wsdl` will look like:

```
WSDL Services available
ContainerService(http://ws.iona.com/container)
ContainerService(http://ws.iona.com/container)
```

WSIL support

The WSIL specification, available at <http://www-128.ibm.com/developerworks/library/specification/ws-wsilspec>, provides a standard way of inspecting a web service and getting the contracts of active web services.

Using the example system above the WSIL document is available from `http://10.1.2.3:2000/inspection.wsil` and has the following content:

```
<?xml version="1.0"?>
<inspection targetNamespace="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
  xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
  xmlns:wsilwsdl="http://schemas.xmlsoap.org/ws/2001/10/inspection/wsdl/">
  <service>
    <description referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
      location="http://10.1.2.3:2000/get_wsdl/opt/IONA/artix/3.0/wsdl/container.wsdl">
      <wsilwsdl:reference>
        <wsilwsdl:referencedService xmlns:ns1="http://ws.iona.com/container">
          ns1:ContainerService
        </wsilwsdl:referencedService>
      </wsilwsdl:reference>
    </description>
  </service>
  <service>
    <description referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
      location="http://10.1.2.3:2000/services/container/ContainerService?wsdl">
      <wsilwsdl:reference>
        <wsilwsdl:referencedService xmlns:ns1="http://ws.iona.com/container">
          ns1:ContainerService
        </wsilwsdl:referencedService>
      </wsilwsdl:reference>
    </description>
  </service>
</inspection>
```

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.

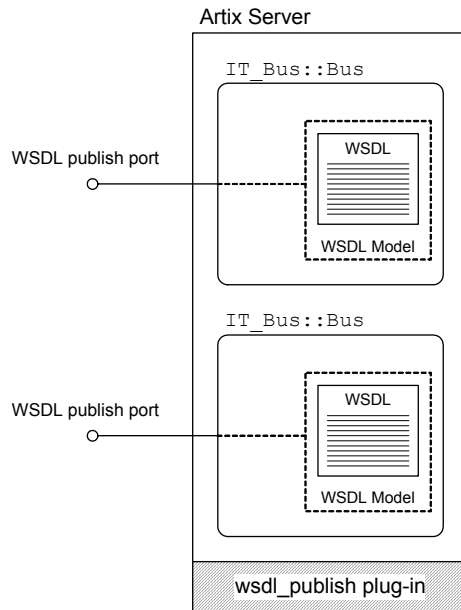
Whether or not you can use the physical part of the WSDL model depends on how the corresponding servant is registered on the server side:

- 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 Servants” on page 64](#).

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 9](#).

Figure 9: *WSDL Publish Plug-In and Multiple Bus Instances*



References to Transient Services

Overview

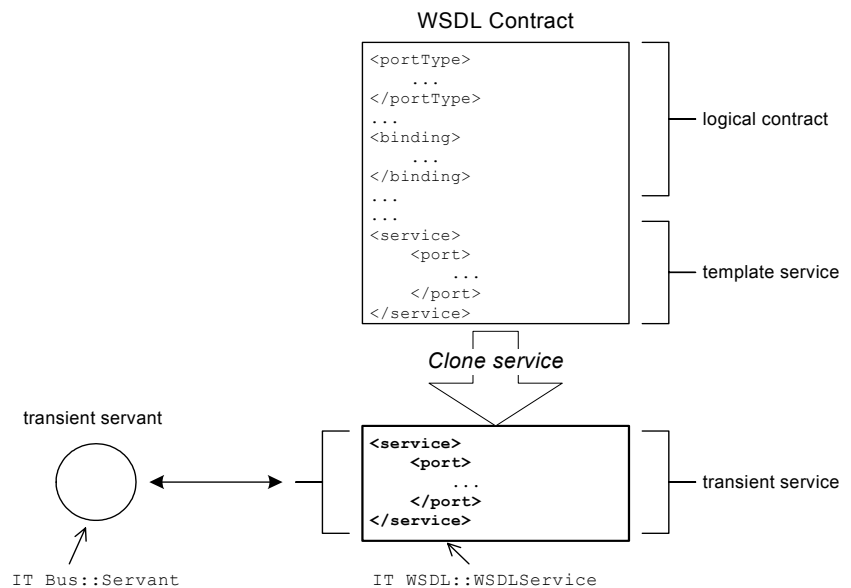
Sometimes you need to be able to define an unlimited number of services, where all of the services are associated with the same port type. For example, a port type could be defined to represent bank account objects. Because each service instance is meant to represent a single user's account, you would need to define an unlimited number of account services. The service must, therefore, be defined as a *transient service*.

A transient service is a dynamically defined service which is created when you call the `IT_Bus::Bus::register_transient_servant()` function. For details, see ["Registering a Transient Servant" on page 72](#).

Creating transient services

[Figure 10](#) gives you an overview of the mechanism that Artix employs to create transient services.

Figure 10: *Cloning a Transient Service from a Template Service*



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).

Cloning a transient service

A transient service is created whenever the application registers a servant as transient, using the `register_transient_service()` function.

To create the new transient service, Artix selects the template service whose service QName and port name match the values specified to `register_transient_service()`. Artix then clones a transient service from the template, making the following changes:

- A unique service QName, obtained by mangling the original template service name, is generated for the transient service.
 - The port address is affected in a transport-dependent manner:
 - ◆ *HTTP transport*—the unique service name is appended to the placeholder URL.
 - ◆ *CORBA and Tunnel transports*—the `ior:` placeholder IOR is replaced by a unique IOR.
-

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 47](#) shows an example of a SOAP service that could be used as a template for cloning transient SOAP services.

Example 47: 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 48](#) shows an example of a CORBA service that could be used as a template for cloning transient CORBA services.

Example 48: 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.

Programming with References

Overview

This section explains how to program with Artix 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.

In this section

This section contains the following subsections:

Bank WSDL Contract	page 117
Creating References	page 126
Resolving References	page 130

Bank WSDL Contract

Overview

This subsection describes the Bank WSDL contract, which demonstrates a typical scenario where Artix references would be used.

The XML Reference type

Artix defines a proprietary XML schema that defines the reference type for use within WSDL contracts. The reference type is `RefPrefix:Reference`, where `RefPrefix` is associated with the following namespace URI:

`http://schemas.iona.com/references`

The references XML schema

The definition of the references schema can be found in the following file:

`ArtixInstallDir/artix/Version/schemas/references.xsd`

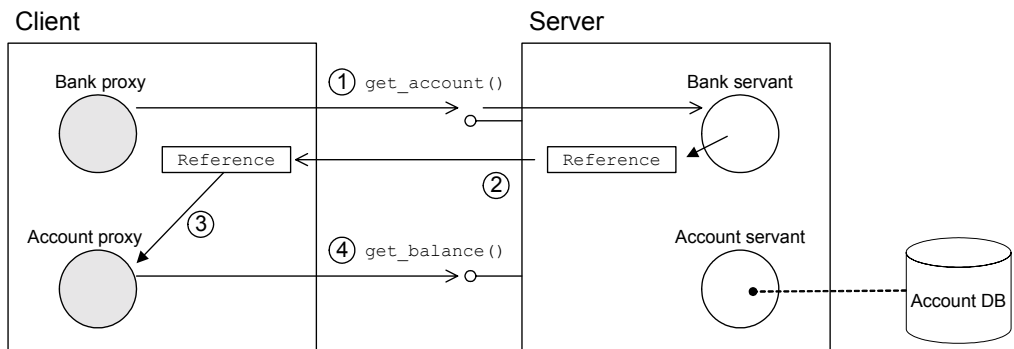
The schema is also available online at:

<http://schemas.iona.com/references/references.xsd>

The Bank example

Figure 11 shows an overview of the Bank example, illustrating how the Bank service uses references to give a client access to a specific account.

Figure 11: 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.

The Bank WSDL contract

[Example 49](#) 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 49: Bank WSDL Contract

```

1 <?xml version="1.0" encoding="UTF-8"?>
  <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/transport/stub"
    xmlns:http="http://schemas.iona.com/transport/http"
    xmlns:http-conf="http://schemas.iona.com/transport/http/configuration"
    xmlns:fixed="http://schemas.iona.com/bindings/fixed"
    xmlns:iiop="http://schemas.iona.com/transport/iiop_tunnel"
    xmlns:corba="http://schemas.iona.com/bindings/corba"

    xmlns:ns1="http://www.iona.com/corba/typemap/BasePortType.idl"
    "
    xmlns:references="http://schemas.iona.com/references"
    xmlns:mq="http://schemas.iona.com/transport/mq"
    xmlns:routing="http://schemas.iona.com/routing"
    xmlns:messaging="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>

```

Example 49: Bank WSDL Contract

```

2      <xsd:import
schemaLocation="../../../../schemas/references.xsd"
namespace="http://schemas.iona.com/references"/>
      <schema elementFormDefault="qualified"
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>
3      <part name="return" type="references:Reference"/>
</message>

<message name="get_account">
  <part name="account_name" type="xsd:string"/>
</message>
4      <part name="return" type="references:Reference"/>
</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">

```

Example 49: Bank WSDL Contract

```

    <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"/>
      <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>

```


Example 49: Bank WSDL Contract

```

    </operation>
</portType>

<binding name="BankBinding" type="tns:Bank">
  <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="list_accounts">
    <soap:operation
      soapAction="http://www.iona.com/bus/demos/bank"
      style="rpc"/>
    <input>
      <soap:body use="literal"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://www.iona.com/bus/demos/bank"/>
    </input>
    <output>
      <soap:body use="literal"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://www.iona.com/bus/demos/bank"/>
    </output>
  </operation>
  <operation name="create_account">
    <soap:operation
      soapAction="http://www.iona.com/bus/demos/bank" style="rpc"/>
    <input>
      <soap:body use="literal"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://www.iona.com/bus/demos/bank"/>
    </input>
    <output>
      <soap:body use="literal"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://www.iona.com/bus/demos/bank"/>
    </output>
  </operation>
  <operation name="get_account">
    <soap:operation
      soapAction="http://www.iona.com/bus/demos/bank" style="rpc"/>
    <input>
      <soap:body use="literal"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://www.iona.com/bus/demos/bank"/>
    </input>
    <output>

```

Example 49: Bank WSDL Contract

```

        <soap:body use="literal"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://www.iona.com/bus/demos/bank"/>
    </output>
</operation>
<operation name="delete_account">
    <soap:operation
soapAction="http://www.iona.com/bus/demos/bank" style="rpc"/>
    <input>
        <soap:body use="literal"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://www.iona.com/bus/demos/bank"/>
    </input>
    <output>
        <soap:body use="literal"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://www.iona.com/bus/demos/bank"/>
    </output>
    </operation>
</binding>

<binding name="AccountBinding" type="tns:Account">
    <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="get_balance">
        <soap:operation
soapAction="http://www.iona.com/bus/demos/bank" style="rpc"/>
        <input>
            <soap:body use="literal"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://www.iona.com/bus/demos/bank"/>
        </input>
        <output>
            <soap:body use="literal"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://www.iona.com/bus/demos/bank"/>
        </output>
    </operation>
    <operation name="deposit">
        <soap:operation
soapAction="http://www.iona.com/bus/demos/bank" style="rpc"/>
        <input>
            <soap:body use="literal"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://www.iona.com/bus/demos/bank"/>

```

Example 49: Bank WSDL Contract

```

        </input>
        <output>
            <soap:body use="literal"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://www.iona.com/bus/demos/bank"/>
        </output>
    </operation>
</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 `references` prefix with the `http://schemas.iona.com/references` namespace URI. This means that the reference type is identified as `references:Reference`.
2. The `xsd:import` imports the `<references:Reference>` type definition from the `references` schema, `references.xsd`. You must edit this line if the `references` schema is stored at a different location relative to the bank WSDL file.

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 `references:Reference` type.
4. The `get_accountResponse` message (which is the `out` parameter of the `get_account` operation) is defined to be of `references:Reference` type.
5. The `create_account` operation defined on the `Bank` port type is defined to return a `references:Reference` type.
6. The `get_account` operation defined on the `Bank` port type is defined to return a `references:Reference` 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.

Creating References

Overview

This subsection describes how to create Artix references, which can be generated on the server side in order to advertise a service's addressing details 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](#).

Factory pattern

References are usually created in the context of a *factory pattern*. This pattern involves at least two kinds of object:

- One type of object, to which the references refer.
- Another type of object, the *factory*, which generates references to the first type.

For example, the Bank is a factory that generates references to Accounts.

Creating a reference from a static servant

[Example 50](#) shows how to create a `BankService` reference from a static servant, `BankImpl`.

Example 50: *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.ionas.com/bus/demos/bank"
    );

1   BankImpl my_bank(bus);
```

Example 50: *Creating a Reference from a Static Servant*

```

2   bus->register_servant(
        my_bank,
        "../wsdl/bank.wsdl",
        service_name,
        "BankPort"
    );
3
4   IT_Bus::Service_var service = bus->get_service(service_name);
    IT_Bus::Reference& bank_reference = service->get_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. The `register_servant()` function registers a static servant instance, taking the following arguments:
 - ◆ Servant instance.
 - ◆ WSDL file location.
 - ◆ Service QName.
 - ◆ Port name (optional).

Note: If the port name argument is omitted, all of the service's ports will be activated.

The return value is an `IT_Bus::Service` object, which references the original `BankService` WSDL service.

3. Call `IT_Bus::Bus::get_service()` to get a pointer to the `Service` object.
4. The `get_reference()` function returns an Artix reference for the service object, `service`.

Creating a reference from a transient servant

Example 51 gives the implementation of the `BankImpl::create_account()`, function which shows how to create an `AccountService` reference from a transient servant, `AccountImpl`.

Example 51: Creating a Reference from a Transient Servant

```

// C++
void
BankImpl::create_account(
    const IT_Bus::String &account_name,
    IT_Bus::Reference &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      );
        Service_var service =
            get_bus()->register_transient_servant(
                *new_account,
                "../wsdl/bank.wsdl",
                AccountImpl::SERVICE_NAME
            );

        // Now put the details for the account into the map so
        // we can retrieve it later.
        //
        AccountDetails details;
        details.m_service = service.release();
        details.m_account = new_account;

        account_iter = m_account_map.insert(
            AccountMap::value_type(account_name, details)
        ).first;
    }

3      account_reference
        = (*account_iter).second.m_service->get_reference()

```


Example 51: *Creating a Reference from a Transient Servant*

```
}

```

The preceding C++ code can be described as follows:

1. This line creates an `AccountImpl` servant instance, which implements the `Account` port type.
2. The `register_transient_servant()` function registers a transient servant instance, taking the following arguments:
 - ◆ Servant instance.
 - ◆ WSDL file location.
 - ◆ Service QName.
 - ◆ Port name (optional).

Note: If the port name argument is omitted, all of the service's ports will be activated.

The return value is an `IT_Bus::Service` object, which references a WSDL service cloned from `AccountService`.

3. The `get_reference()` function returns an Artix reference for the account service object.

Creating a reference from a WSDL contract

You can create a reference directly from an `IT_Bus::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);
IT_Bus::Reference result;

bus->populate_endpoint_reference(
    *wsdl_service,
    result
);
```

Resolving References

Overview

To a client, an `IT_Bus::Reference` 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 a special constructor to initialize the proxy from an `IT_Bus::Reference` object. For example, the `AccountClient` proxy class includes the following constructor:

```
// C++
AccountClient(const IT_Bus::Reference&);
```

The data to initialize the `AccountClient` object is obtained partly from the `IT_Bus::Reference` object (service and port details) and partly from the WSDL contract (port type and binding details).

Client example

[Example 52](#) 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 52: *Client Using an Account Reference*

```
// C++
...
BankClient bankclient;

// 1. Retrieve an account reference from the remote Bank object.
IT_Bus::Reference 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);
```

Callbacks

An Artix callback is an implementation pattern, where a client implements a WSDL service (thus exhibiting hybrid client/server behavior). Because the server initially does not know about the client's service, the client must transmit a callback reference to the server (that is, register the callback). The server is then able to call back on the client's service at a later time.

In this chapter

This chapter discusses the following topics:

Overview of Artix Callbacks	page 132
Routing and Callbacks	page 134
Callback WSDL Contract	page 138
Client Implementation	page 141
Server Implementation	page 145

Overview of Artix Callbacks

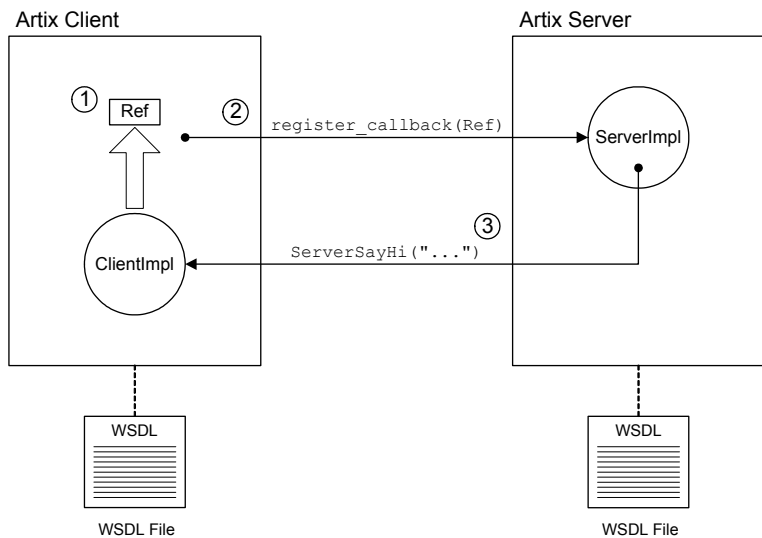
Overview

The callback example described in this section is based on Artix callback demonstration, which is located in the following directory:

`ArtixInstallDir/artix/Version/demos/callbacks/basic_callback`

Callbacks rely, essentially, on Artix 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 12 illustrates how this process works.

Figure 12: Overview of the Callback Demonstration



Callback steps

[Figure 12 on page 132](#) shows the callback proceeding according to the following steps:

1. After the basic initialization steps, including registration of the `ClientImpl` servant and `ClientService` 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 `register_callback()` 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 `ClientImpl` 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.

Threading

By default, both the client and the server allocate a pool of threads to process incoming requests (see [“Multi-Threading” on page 80](#)). Hence, the client’s callback is called in a pool thread, not in a user thread, and the callback implementation must be synchronized appropriately.

One of the positive side effects of this policy is that the callback scenario shown in [Figure 12 on page 132](#) is *not* subject to deadlock.

Note: In the current example, it is also significant that the client service is activated as soon as it is registered. Otherwise the code shown in [Example 54 on page 141](#) would lead to deadlock.

Routing and Callbacks

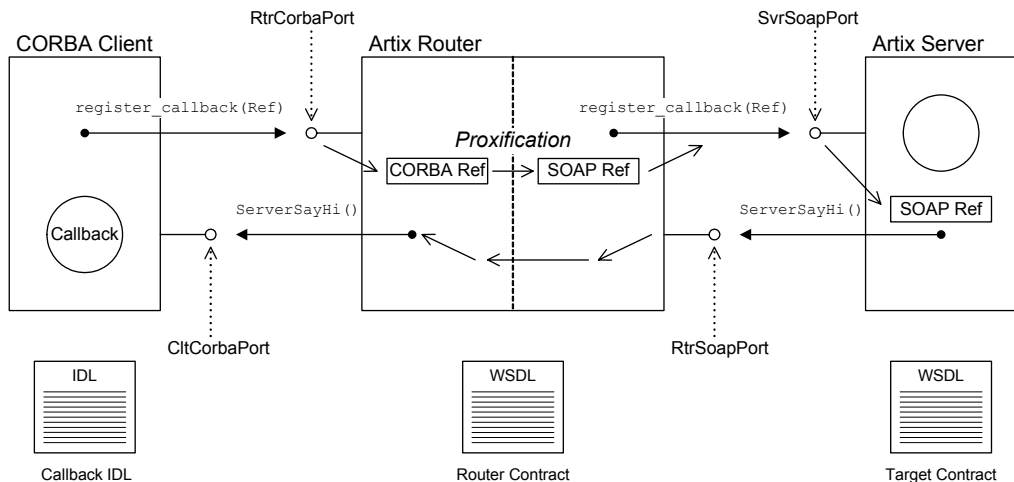
Overview

Callbacks are fully compatible with Artix routers. Reference that passes 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 some protocol routing, proxification is disabled by default.

For example, consider the callback routing scenario shown in [Figure 13](#). 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 13: Overview of a Callback Routing Scenario



Contracts

The applications in [Figure 13](#) are associated with 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.

You can generate a router contract using the Artix Designer GUI tool. 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 13 on page 134](#), 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 `register_callback()` 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 `ClcCorbaPort` 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, an Artix SOAP reference).

The proxification process is of key importance to Artix callbacks. If the router in [Figure 13 on page 134](#) did not proxify `register_callback()`'s reference argument, it would be impossible for the server to call back on the client. The server can communicate *only* with SOAP/HTTP endpoints, not with IIOP endpoints.

In [Figure 13 on page 134](#), the router proxifies the callback reference as follows:

1. When the `register_callback()` 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 `register_callback()` 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.

Callback WSDL Contract

Overview

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 53](#) shows the WSDL contract used for the callback demonstration.

Example 53: Example Callback WSDL Contract

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="callback_demo"
  targetNamespace="http://www.iona.com/callback"
  xmlns="http://schemas.xmlsoap.org/wsdl/"

  xmlns:http-conf="http://schemas.iona.com/transport/http/conf
  igation"
  xmlns:references="http://schemas.iona.com/references"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.iona.com/callback"
  xmlns:wSDL="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <xsd:import
      namespace="http://schemas.iona.com/references"
      schemaLocation="../../../../schemas/references.xsd"/>
    <schema targetNamespace="http://www.iona.com/callback"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:wSDL="http://schemas.xmlsoap.org/wsdl/"
      <element name="register_callback.c"
      type="references:Reference"/>
    </schema>
  </types>
  <message name="ServerSayHi">
    <part name="param" type="xsd:string"/>
  </message>
  <message name="register_callback">
    <part element="tns:register_callback.c" name="c"/>
  </message>
```

Example 53: *Example Callback WSDL Contract*

```

1  <portType name="ClientPortType">
    <operation name="ServerSayHi">
        <input message="tns:ServerSayHi" name="ServerSayHi"/>
    </operation>
</portType>

2  <portType name="ServerPortType">
    <operation name="register_callback">
        <input message="tns:register_callback"
            name="register_callback"/>
    </operation>
</portType>
...
<service name="ClientService">
    <port binding="tns:ClientPortType_SOAPBinding"
        name="ClientPort">
3      <soap:address location="http://localhost:0"/>
        <http-conf:client/>
        <http-conf:server/>
    </port>
</service>

    <service name="ServerService">
        <port binding="tns:ServerPortType_SOAPBinding"
            name="SOAPPort">
4      <soap:address location="http://SvrHost:SvrPort"/>
        <http-conf:client/>
        <http-conf:server/>
    </port>
</service>
</definitions>

```

1. The `ClientPortType` 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:
 - ◆ `register_callback` operation—takes a single Artix 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

Overview

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.](#)
- [ClientImpl servant class.](#)

Client main function

[Example 54](#) shows the code for the callback client main function, which instantiates and registers a `ClientImpl` servant before calling on the remote server to register the callback.

Example 54: *Callback Client Main Function*

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

#include "ServerClient.h"
#include "ClientImpl.h"

IT_USING_NAMESPACE_STD

using namespace DemosCallback;
using namespace IT_Bus;

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

    try
    {
        cout << "Initializing Bus." << endl;
        Bus_var bus = IT_Bus::init(argc, argv);
    }
}
```

Example 54: *Callback Client Main Function*

```

1 ClientImpl servant(bus);
  cout << "Activating Service on Bus" << endl;
2 QName service_qname(
    "", "ClientService", "http://www.iona.com/callback"
  );

3 bus->register_servant(
    servant,
    "../../etc/callback.wsdl",
    service_qname
  );

  IT_Bus::Service_var service =
    bus->get_service(service_qname);

4 IT_Bus::Reference & client_ref =
    service->get_reference();
  ServerClient sc("../../etc/callback.wsdl");
5 sc.register_callback(client_ref);

  cout << "Callback Ready." << endl;
6 while (! ClientImpl::callback_received ) {
    // ... do something useful!
    IT_CurrentThread::sleep(100); // 100 ms
  }

  bus->shutdown(true);
  cout << "Done." << endl;
}
catch(IT_Bus::Exception& e)
{
  cout << endl << "Error : Unexpected error occurred!"
    << endl << e.message()
    << endl;
  return -1;
}
return 0;
}

```

The preceding code example can be explained as follows:

1. The `ClientImpl` servant class implements the `ClientPortType` port type. The `ClientImpl` 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="ClientService">` element in [Example 53 on page 138](#).
3. Register the callback servant with the Bus, thereby activating the `ClientService` service. From this point on, the `ClientService` 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_reference()`.
5. This line invokes the `register_callback()` operation on the remote server, passing in the reference to the client callback object. From this point on, the server could invoke an operation on the callback.
6. The main thread remains in a `while` loop until a flag, `ClientImpl::callback_received`, is set to true.

ClientImpl servant class

[Example 55](#) shows the implementation of the `ClientImpl` servant class, which is responsible for receiving the `ClientImpl::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 55: ClientImpl Servant Class Implementation

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

IT_USING_NAMESPACE_STD
using namespace DemosCallback;

ClientImpl::ClientImpl (
    IT_Bus::Bus_ptr bus
) : DemosCallback::ClientServer (bus)
{
    // complete
```

Example 55: *ClientImpl Servant Class Implementation (Continued)*

```
}

ClientImpl::~ClientImpl()
{
    // Complete
}

void
ClientImpl::ServerSayHi(
    const IT_Bus::String & param
) IT_THROW_DECL((IT_Bus::Exception))
{
    cout <<"ClientImpl::ServerSayHi() called"<<endl;
    cout << param <<endl;
    cout <<"ClientImpl::ServerSayHi() ended"<<endl;

    callback_received = true;
}
}
```

Server Implementation

Overview

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 56](#) 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 `register_callback` operation.

Example 56: Server Main Function

```
// C++
#include <it_bus/bus.h>
#include <it_bus/service.h>
#include <it_bus/exception.h>
#include <it_bus/fault_exception.h>
#include <it_bus/file_output_stream.h>

#include "ServerImpl.h"

IT_USING_NAMESPACE_STD

using namespace IT_Bus;
using namespace DemosCallback;

int
main(int argc, char* argv[])
{
    try
    {
        cout << "Initializing Bus." << endl;
        IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);

        ServerImpl servant(bus);
        IT_Bus::QName service_qname(
```

1
2

Example 56: Server Main Function (Continued)

```

        "", "ServerService", "http://www.ionas.com/callback"
    );
3   bus->register_servant(
        servant,
        "../../etc/callback.wsdl",
        service_qname
    );

4   cout << "Service Ready." << endl;
    IT_Bus::run();

    bus->shutdown(true);
    cout << "Done." << endl;
}
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 `ServerImpl` servant class implements the `ServerPortType` port type, which supports the `register_callback` operation.
2. The `service_qname` refers to the `<service name="ServerService">` element in [Example 53 on page 138](#).
3. Register the `ServerImpl` servant with the Bus, thereby activating the `ServerService` service.
4. Call the blocking `IT_Bus::run()` function to allow the server application to process incoming requests.

ServerPortType implementation

Example 57 shows the implementation of the `ServerImpl` servant class. There is just one WSDL operation, `register_callback()`, to implement in this class.

Example 57: ServerImpl Servant Class Implementation

```

// C++
#include "ServerImpl.h"
#include <it_cal/cal.h>

IT_USING_NAMESPACE_STD
using namespace DemosCallback;

ServerImpl::ServerImpl(IT_Bus::Bus_ptr bus) :
    DemosCallback::ServerServer(bus)
{
    // Complete
}

ServerImpl::~ServerImpl()
{
    // Complete
}

void
ServerImpl::register_callback(
1     const IT_Bus::Reference & c
) IT_THROW_DECL((IT_Bus::Exception))
{
    cout << "ServerImpl::register_callback(): called"<< endl;
    cout << "Calling Back to client" << endl;

    try
    {
2         ClientClient cc(c);
3         cc.ServerSayHi("Server says hi to client");
    }
    catch(IT_Bus::Exception& e)
    {
        cout << "Caught Unexpected Exception: " << e.message() <<
endl;
    }
    catch (...)
    {
        cout << "Unknown exception" << endl;
    }
}

```

Example 57: *ServerImpl Servant Class Implementation*

```
    }  
    cout << "Finished callback to client" << endl;  
    cout << "ServerImpl::register_callback(): returning"<< endl;  
}
```

The preceding code example can be explained as follows:

1. The `register_callback()` function takes a reference argument, which should be a reference to a callback object.
2. This line creates a client proxy, `cc`, for the `ClientPortType` port type and initializes it with the callback reference, `c`. The reference, `c`, encapsulates details of the `ClientService` service.
3. This line invokes the `ServerSayHi()` callback on the client.

This example, where the callback is invoked within the body of `register_callback()`, 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.

The Artix Locator

The Artix locator is a central repository for storing references to Artix endpoints. If you set up your Artix servers to register their endpoints with the locator, you can code your clients to open server connections by retrieving endpoint references from the locator.

In this chapter

This chapter discusses the following topics:

Overview of the Locator	page 150
Locator WSDL	page 153
Registering Endpoints with the Locator	page 159
Reading a Reference from the Locator	page 161

Overview of the Locator

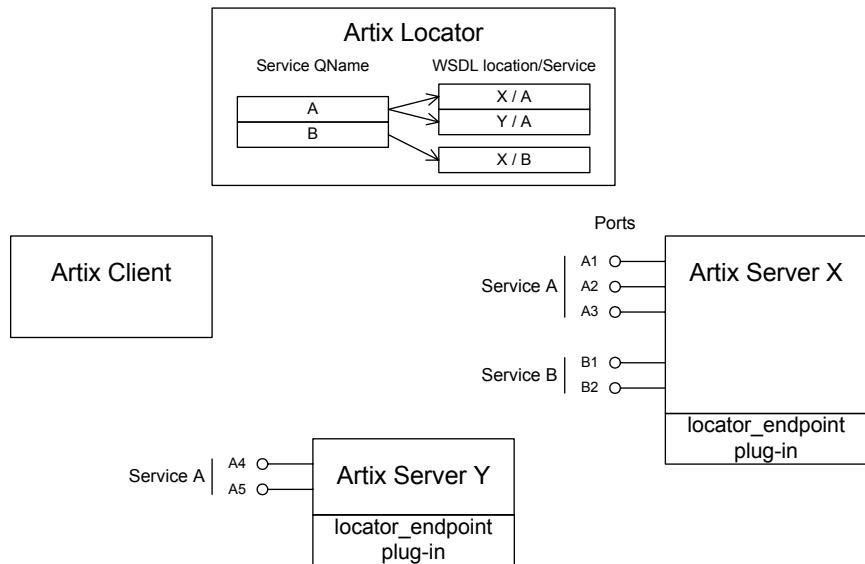
Overview

The Artix locator is a service which can optionally be deployed for the following purposes:

- *Repository of endpoint references*—endpoint references stored in the locator enable clients to establish connections to Artix services.
- *Load balancing*—if multiple service instances (identified by a WSDL location and service QName) are registered against a single service QName, the locator load balances over the different service instances using a round-robin algorithm.

Figure 14 gives a general overview of the locator architecture.

Figure 14: Artix Locator Overview



Locator demonstration

The locator demonstration, which forms the basis of the examples in this section, is located in the following directory:

```
ArtixInstallDir/artix/Version/demos/advanced/locator
```

Locator service

There are two options for deploying the locator service, as follows:

- *Standalone deployment*—the locator is deployed as an independent server process (as shown in [Figure 14](#)). This approach is described in detail in the “Using the Artix Locator” chapter from the *Deploying and Managing Artix Solutions* document. Sample source code for such a standalone locator service is provided in the `demos/advanced/locator` demonstration.
 - *Embedded deployment*—the locator is deployed by embedding it within another Artix server process. This approach is possible because the locator is implemented as a plug-in, which can be loaded into any Artix application.
-

Endpoint definition

An Artix *endpoint* is a particular WSDL service (identified by a service QName) in a particular `IT_Bus:Bus` instance (identified by a WSDL location URL). Hence, it is possible to have endpoints with the same service type and service QName, as long as they are registered with different Bus instances. A WSDL location URL and a service QName together identify an endpoint.

Registering endpoints

A server registers its endpoints with the locator in order to make them accessible to Artix clients. When a server registers an endpoint in the locator, it creates an entry in the locator that associates a service QName with an Artix reference for that endpoint.

Looking up references

An Artix client looks up a reference in the locator in order to find an endpoint associated with a particular service. After retrieving the reference from the locator, the client can then establish a remote connection to the relevant server by instantiating a client proxy object. This procedure is independent of the type of binding or transport protocol.

Load balancing with the locator

If multiple endpoints are registered against a single service QName in the locator, the locator will employ a round-robin algorithm to pick one of the endpoints. Hence, the locator effectively *load balances* a service over all of its associated endpoints.

For example, [Figure 14 on page 150](#) shows the `Service A` QName with two endpoints registered against it:

- WSDL location X / Service A
- WSDL location Y / Service A

When the Artix client looks up a reference for `Service A`, it obtains a reference to whichever endpoint is next in the sequence.

Locator WSDL

Overview

The locator WSDL contract, `locator.wsdl`, defines the public interface of the locator through which the service can be accessed either locally or remotely. A copy of the locator WSDL contract is stored in the following location:

```
ArtixInstallDir/artix/Version/wsdl/locator.wsdl
```

This section shows extracts from the locator WSDL that are relevant to normal user applications. The following aspects of the locator WSDL are described here:

- [Binding and protocol](#)
- [WSDL contract](#)
- [C++ mapping](#)

Binding and protocol

The locator service is normally accessed through the SOAP binding and over the HTTP protocol.

WSDL contract

[Example 58](#) shows an extract from the locator WSDL contract that focuses on the aspects of the contract relevant to an Artix application programmer. There is just one WSDL operation, `lookup_endpoint`, that an Artix client typically needs to call.

Example 58: Extract from the Locator WSDL Contract

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:ref="http://schemas.iona.com/references"
  xmlns:ls="http://ws.iona.com/locator"
  targetNamespace="http://ws.iona.com/locator">
  <types>
    <xs:schema attributeFormDefault="unqualified"
      elementFormDefault="unqualified"
      targetNamespace="http://schemas.iona.com/references">
      <xs:complexType name="ReferencePort">
        <xs:sequence>
```

Example 58: *Extract from the Locator WSDL Contract*

```

        <xs:any maxOccurs="unbounded" minOccurs="0"
            namespace="##other"
            processContents="lax"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:NCName"
        use="required"/>
    <xs:attribute name="binding" type="xs:QName"
        use="required"/>
    </xs:complexType>
</xs:schema>
<xs:schema targetNamespace="http://ws.iona.com/locator">
1   <xs:import
    namespace="http://schemas.iona.com/references"/>
    ...
2   <xs:element name="lookupEndpoint">
    <xs:complexType>
    <xs:sequence>
        <xs:element name="service_qname"
            type="xs:QName"/>
    </xs:sequence>
    </xs:complexType>
</xs:element>
3   <xs:element name="lookupEndpointResponse">
    <xs:complexType>
    <xs:sequence>
        <xs:element name="service_endpoint"
            type="ref:Reference"/>
    </xs:sequence>
    </xs:complexType>
</xs:element>
    <xs:complexType
name="EndpointNotExistFaultException">
    <xs:sequence>
        <xs:element name="error" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
4   <xs:element name="EndpointNotExistFault"
        type="ls:EndpointNotExistFaultException"/>
    </xs:schema>
</types>
...
<message name="lookupEndpointInput">
    <part name="parameters" element="ls:lookupEndpoint"/>
</message>
<message name="lookupEndpointOutput">

```

Example 58: *Extract from the Locator WSDL Contract*

```

    <part name="parameters"
    element="ls:lookupEndpointResponse"/>
  </message>
  <message name="endpointNotExistFault">
    <part name="parameters"
    element="ls:EndpointNotExistFault"/>
  </message>

5  <portType name="LocatorService">
    ...
6  <operation name="lookup_endpoint">
    <input message="ls:lookupEndpointInput"/>
    <output message="ls:lookupEndpointOutput"/>
    <fault name="fault"
    message="ls:endpointNotExistFault"/>
  </operation>
7  <operation name="list_endpoints">
    <input message="ls:listEndpointInput"/>
    <output message="ls:listEndpointOutput"/>
  </operation>
</portType>
<binding name="LocatorServiceBinding"
    type="ls:LocatorService">
    ...
</binding>
<service name="LocatorService">
  <port name="LocatorServicePort"
    binding="ls:LocatorServiceBinding">
    <soap:address
8  location="http://localhost:0/services/locator/LocatorService"/>
    </port>
  </service>
</definitions>

```

The preceding locator WSDL extract can be explained as follows:

1. This line imports the namespace for the references schema. The references schema itself is embedded in the WSDL contract, just preceding this schema.
2. The `lookupEndpoint` type is the input parameter type for the `lookup_endpoint` operation. It contains just the QName (qualified name) of a particular WSDL service.

3. The `lookupEndpointResponse` type is the output parameter type for the `lookup_endpoint` operation. It contains an Artix reference for the specified service. If more than one endpoint is registered against a particular service name, the locator picks one of the endpoints using a round-robin algorithm.
4. The `EndpointNotExist` fault would be thrown if the `lookup_endpoint` operation fails to find an endpoint registered against the requested service type.
5. The `LocatorService` port type defines the public interface of the Artix locator service.
6. The `lookup_endpoint` operation is called by Artix clients to retrieve endpoint references.
7. The `list_endpoints` operation returns a list of all endpoints stored in the locator.
8. The SOAP `location` attribute specifies the host and IP port for the locator service. You must edit the `location` attribute, specifying a specific host name and a fixed IP port.

Note: The default location setting, `localhost:0`, is *not* suitable for deployment. When the port setting is `0`, Artix allocates the locator port dynamically. This would make the locator service unusable, because the IP port would be different each time it starts up.

C++ mapping

[Example 59](#) shows an extract from the C++ mapping of the `LocatorService` port type. This extract shows only the `lookup_endpoint` WSDL operation—the other WSDL operations in this class are normally not needed by user applications.

Example 59: C++ Mapping of the `LocatorService` Port Type

```
// C++
#include "LocatorService.h"
#include <it_bus/service.h>
#include <it_bus/bus.h>
#include <it_bus/reference.h>
#include <it_bus/types.h>
#include <it_bus/operation.h>
```

Example 59: C++ Mapping of the LocatorService Port Type

```

namespace IT_Bus_Services
{
    namespace IT_Locator {
        class LocatorServiceClient
            : public LocatorService, public IT_Bus::ClientProxyBase
        {
        public:
            LocatorServiceClient(
                IT_Bus::Bus_ptr bus = 0
            );

            LocatorServiceClient(
                const IT_Bus::String & wsdl,
                IT_Bus::Bus_ptr bus = 0
            );

            LocatorServiceClient(
                const IT_Bus::String & wsdl,
                const IT_Bus::QName & service_name,
                const IT_Bus::String & port_name,
                IT_Bus::Bus_ptr bus = 0
            );

            LocatorServiceClient(
                IT_Bus::Reference & reference,
                IT_Bus::Bus_ptr bus = 0
            );

            ~LocatorServiceClient();
            ...
            virtual void
            lookup_endpoint(
                const IT_Bus::QName &service_qname,
                IT_Bus::Reference &service_endpoint
            ) IT_THROW_DECL((IT_Bus::Exception)) = 0;

            virtual void
            list_endpoints(
                IT_Bus::ElementListT<IT_Bus_Services::IT_Locator::endpoint>
                &endpoint
            ) IT_THROW_DECL((IT_Bus::Exception)) = 0;
        };
    };
};

```

The value returned by `list_endpoint` is an element list of `IT_Bus_Services::IT_Locator::endpoint` objects, which are defined by the C++ class shown in [Example 60](#).

Example 60: *The `IT_Bus_Services::IT_Locator::endpoint` Class*

```
// C++
namespace IT_Bus_Services {
    namespace IT_Locator {
        class endpoint : public IT_Bus::SequenceComplexType
        {
        public:
            static const IT_Bus::QName&
            get_static_type();

            endpoint();
            endpoint(const endpoint & copy);
            virtual ~endpoint();
            ...
            IT_Bus::String & getnode_id();
            const IT_Bus::String & getnode_id() const;

            void
            setnode_id(const IT_Bus::String & val);

            IT_Bus::Reference & getendpoint_reference();

            const IT_Bus::Reference &
            getendpoint_reference() const;

            void
            setendpoint_reference(const IT_Bus::Reference & val);
            ...
        };
    };
};
```

Registering Endpoints with the Locator

Overview

To register a server's endpoints with the locator, you must configure the server to load a specific set of plug-ins. The server will then, by default, automatically register every endpoint (that is, service/port combination) created on the server side. If you need more control over endpoint registration, Artix provides a filtering feature (see *Developing and Managing Artix Solutions* for details).

There is currently no programming API for registering endpoints explicitly.

Configuring a server to register endpoints

A server that is to register its endpoints with the locator should be configured to include the `locator_endpoint` plug-in, as shown in the following `demo.locator.server` configuration scope from `artix.cfg`:

```
# Artix Configuration File
...
demo {
    locator {
        server
        {
            bus:initial_contract:url:locator = "locator.wsdl";
            orb_plugins = ["xmlfile_log_stream", ..., "soap",
"at_http", "locator_endpoint"];
        };
    };
    ...
};
```

When running the server, remember to select the appropriate configuration scope by passing it as the `-ORBname` command-line parameter. For example, the preceding configuration would be picked up by a `MyArtixServer` executable, if the server is launched with the following command:

```
MyArtixServer -ORBname demo.locator.server
```

`bus:initial_contract:url:locator` configuration variable

The `bus:initial_contract:url:locator` configuration variable specifies the location of the locator WSDL file, `locator.wsdl`.

References

For more details about configuring a server to register endpoints, see the following references:

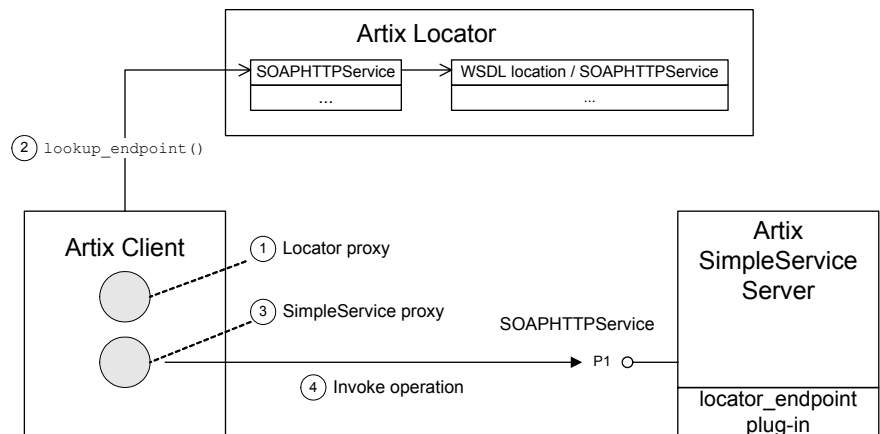
- “Using the Artix Locator” chapter from the *Developing and Managing Artix Solutions* document.
- The Artix `locator` demonstration in `artix/Version/demos/advanced/locator`.

Reading a Reference from the Locator

Overview

After the target server (in this example, the `SimpleService` server) has started up and registered its endpoints with the locator, an Artix client can then bootstrap a connection to the target server by reading one of its endpoint references from the locator. [Figure 15](#) shows an outline of how a client bootstraps a connection in this way.

Figure 15: Steps to Read a Reference from the Locator



Programming steps

The main programming steps needed to read a reference from the locator, as shown in [Figure 15](#), are as follows:

1. Construct a locator service proxy.
2. Use the locator proxy to invoke the `lookup_reference` operation.
3. Use the reference returned from `lookup_reference` to construct a `SimpleService` proxy.
4. Invoke an operation using the `SimpleService` proxy.

Example

Example 61 shows an example of the code for an Artix client that retrieves a reference to a `SimpleService` service from the Artix locator.

Example 61: *Example of Reading a Reference from the Locator Service*

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

#include "SimpleServiceClient.h"
#include "LocatorServiceClient.h"

IT_USING_NAMESPACE_STD
using namespace IT_Bus;
using namespace IT_Bus_Services::IT_Locator;
using namespace SimpleServiceNS;

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

    try
    {
        int my_argc = 2;
        const char * my_argv [] = {
            "-ORBname",
            "demo.locator.client"
        };

1         IT_Bus::Bus_var bus = IT_Bus::init(
                my_argc,
                (char **)my_argv
            );

2         QName ls_service_name(
            "", "LocatorService", "http://ws.iona.com/locator"
        );

3         QName sh_service_name(
            "", "SOAPHTTPService", "http://www.iona.com/bus/tests"
        );
```

Example 61: *Example of Reading a Reference from the Locator Service*

```

// 1. Construct a locator service proxy
IT_Bus::Reference locator_ref;
4 bus->resolve_initial_reference(
    ls_service_name,
    locator_ref
);
LocatorServiceClient locator_client(locator_ref);

// 2. Invoke on locator
IT_Bus::Reference endpoint;
5 locator_client.lookup_endpoint(
    sh_service_name,
    endpoint
);

// 3. Construct a new proxy to your target service with
// the result from the locator
6 SimpleServiceClient sh_simple_client(endpoint);

// 4. Use your new proxy
7 String sh_my_greeting("SOAPHTTP ENDPOINT GREETING");
String result;
sh_simple_client.say_hello(sh_my_greeting, result);
cout << "say_hello method returned: " << result << endl;
}
catch(IT_Bus::Exception& e)
{
    cout << endl << "Caught Unexpected Exception: "
        << endl << e.Message()
        << endl;
    return -1;
}
return 0;
}

```

The preceding C++ example can be explained as follows:

1. You should ensure that the client picks up the correct configuration by passing the appropriate value of the `-ORBname` parameter. In this example, the `-ORBname` parameter is hard-coded, but you might prefer to take this parameter from the command line instead.
2. This line constructs a qualified name, `ls_service_name`, that identifies the `<service name="LocatorService">` tag from the locator WSDL. See the listing of the locator WSDL in [Example 58 on page 153](#).
3. This line constructs a qualified name, `sh_service_name`, that identifies the `SOAPHTTPService` service from the `SimpleService` WSDL.
4. Use the Artix `resolve_initial_reference()` function to obtain a reference to the locator service. Artix implicitly finds the locator service reference using the bootstrapping service.
5. The `lookup_endpoint()` operation is invoked on the locator to find an endpoint of `SOAPHTTPService` type (specified in the `sh_input` parameter).

Note: If there is more than one WSDL port registered for the `SOAPHTTPService` server, the locator service employs a round-robin algorithm to choose one of the ports to use as the returned endpoint.

6. Construct the `SimpleServiceClient` proxy by passing in the `endpoint` reference.
7. You can now use the simple client proxy to make invocations on the remote Artix server.

Using Sessions in Artix

The Artix Session Manager helps you manage service resources.

Note: The session manager is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports the session manager.

In this chapter

This chapter discusses the following topics:

Introduction to Session Management in Artix	page 166
Registering a Server with the Session Manager	page 169
Working with Sessions	page 172

Introduction to Session Management in Artix

Overview

The Artix session manager is a group of ART plug-ins that work together to provide you control over the number of concurrent clients accessing a group of services and how long each client can use the services in the group before having to check back with the session manager. The two main session manager plug-ins are:

Session Manager Service Plug-in (`session_manager_service`) is the central service plug-in. It accepts and tracks service registration, hands out session to clients, and accepts or denies session renewal.

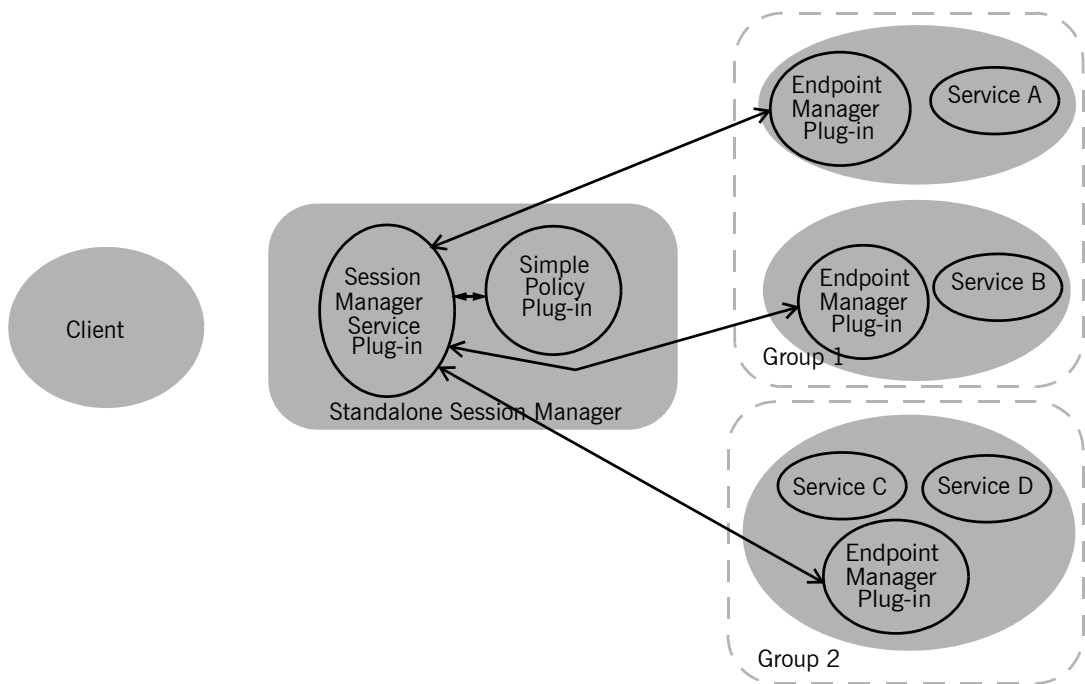
Session Manager Endpoint Plug-in (`session_endpoint_manager`) is the portion of the session manager that resides in a registered service. It registers its location with the service plug-in and accepts or rejects client requests based on the validity of their session headers.

The session manager also has a pluggable policy callback mechanism that allows you to implement your own session management policies. Artix session manager includes a simple policy callback plug-in, `sm_simple_policy`, that provides control over the allowable duration for a session and the maximum number of concurrent sessions allowed for each group.

How do the plug-ins interact?

Figure 16 shows a diagram of how the session manager plug-ins are deployed in an Artix System. As you can see the session manager service plug-in and the policy callback plug-in are both deployed into the same process. While in this example, they are deployed into a standalone service, they can be deployed in any Artix process. The session manager service plug-in and the policy plug-in interact to ensure that the session manager does not hand out sessions that violate the policies established by the policy plug-in.

Figure 16: *The Session Manager Plug-ins*



The endpoint manager plug-ins are deployed into the server processes which contain session managed services. A process can host two services, like Service C and Service D in Figure 16, but the process will have only one endpoint manager. The endpoint manager plug-ins are in constant communication with the session manager service plug-in to report on

endpoint health, to receive information on new sessions that have been granted to the managed services, and to check on the health of the session manager service.

What are sessions?

The session manager controls access to services by handing out *sessions* to clients who request access to the services. A session is a pass that provides access to the services in a specific group for a specific time.

For example if a client application wants to use the services in the `SM_Demo` group, it would ask the session manager for a session with the `SM_Demo` group. The session manager would then check and see if the `SM_Demo` group had an available session, and if so it would return a session id and the list of `SM_Demo` service references to the client. The session manager would then notify the endpoint managers in the `SM_Demo` group that a new session had been issued, the new session's id, and the duration for which the session is valid. When the client then makes requests on the services in the `SM_Demo` group, it must include the session information as part of the request. The endpoint manager for the services then check the session information to ensure it is valid. If it is, the request is accepted. If it is not, the request is rejected.

If the client wants to continue using the `SM_Demo` services beyond the duration of its lease, the client will have to ask the session manager to renew its session before the session expires. Once a client's session has expired, it will have to request a new one.

What are groups?

The Artix session manager does not pass out sessions for each individual service that is registered with it. Instead, services are registered as part of a *group*, and sessions are handed out for the group. A group is a collection of services that are managed as one unit by the session manager. While the session manager does not specify that the services in a group be related, it is recommended that the endpoints have some relationship.

A service's group affiliation is controlled by the configuration scope under which it is run. To change a service's group, you edit the value for `plugins:session_endpoint_manager:default_group` in the process' configuration scope. For more information on Artix configuration see *Deploying and Managing Artix Solutions*.

Registering a Server with the Session Manager

Overview

Services that wish to be managed by the session manager must register with a running session manager. To do this the servers instantiating these services must load the session manager endpoint plug-in and properly configure themselves. They do not require any special application code.

Once registered with a session manager, the services will only accept requests containing a valid session header. All clients wishing to access the services must be written to support session managed services.

Configuring the server

Any server hosting services that are to be managed by the session manager must load the following plug-ins in addition to the transport and payload plug-ins it requires:

- `soap`
- `at_http`
- `session_endpoint_manager`

`session_endpoint_manager` allows the server to register with a running session manager.

The server's configuration also needs to set the following configuration variables:

`bus:initial_contract:url:sessionmanager` points to the contract describing the contact information for the session manager that will be managing the services.

`plugins:session_endpoint_manager:default_group` specifies the default group name for the services instantiated by the server.

[Example 62](#) shows the configuration scope of a server that hosts services managed by the session manager.

Example 62: Server Configuration Scope

```

demos {
  session_management {
    server {
      orb_plugins = ["xmlfile_log_stream", "session_endpoint_manager"];

      # This is the WSDL File that the Session Endpoint Manager used to contact the
      # Session Manager Service.
      bus:initial_contract:url:sessionmanager = "../etc/session-manager.wsdl";
      plugins:session_endpoint_manager:default_group = "SM_Demo";
    };
  };
};

```

A server loaded into the `demos.session_management.server` configuration scope will be managed by the session manager at the location specified in `session-manager.wsdl`, its endpoint manager will come up at the address specified in `session-manager.wsdl`, and by default all services instantiated by the server will belong to the session manager group `SM_Demo`.

For more information on Artix configuration see [Deploying and Managing Artix Solutions](#).

You also need to configure the port on which the endpoint manager will run. To do this you modify `session-manager.wsdl`, provided in the `wsdl` folder of your Artix installation, to specify the HTTP address at which the endpoint manager will be available. Using any text editor, open `session-manager.wsdl` and edit the `<soap:address>` entry for the `SessionEndpointManagerService` to specify the proper address.

[Example 63](#) shows a modified session manager contract entry. The highlighted part has been modified to point to the desired address.

Example 63: Endpoint Manager Address

```

<service name="SessionEndpointManagerService">
  <port name="SessionEndpointManagerPort" binding="sm:SessionEndpointManagerBinding">
    <soap:address
      location="http://localhost:8080/services/sessionManagement/sessionEndpointManager"/>
  </port>
</service>

```

In the server's configuration scope specify the endpoint manager plug-in to read the correct Artix contract for the endpoint manager by setting `bus:initial_contract:url:sessionmanager` to point to the copy of `session-manager.wsdl` containing the address for this instance of the endpoint manager.

Registration

Once a properly configured server starts up, it automatically registers with the session manager specified by the contract pointed to by `bus:initial_contract:url:sessionmanager`.

Working with Sessions

Overview

Clients wishing to make requests from session managed services must be designed explicitly to interact with the Artix session manager and pass session headers to the session managed services.

Demonstration code

The examples in this section are based on the demonstration code located in the following directory:

```
ArtixInstallDir/artix/Version/demos/advanced/session_management
```

Implementing a session client

There are eight steps a client takes when making requests on a session managed service. They are:

1. **Instantiate** a proxy for the session management service.
2. **Start** a session for the desired service's group using the session manager proxy.
3. **Obtain** the list of endpoints available in the group.
4. **Create** a service proxy from one of the endpoints in the group.
5. **Build** a session header to pass to the service.
6. **Invoke** requests on the endpoint using the proxy.
7. **Renew** the session as needed.
8. **End** the session using the session manager proxy when finished with the services.

Instantiating a session manager proxy

Before a client can request a session from the session manager, it must create a proxy to forward requests to the running session manager. To do this the client creates an instance of `SessionManagerClient` using the session manager's contract name, `session-manager.wsdl`.

[Example 64](#) shows how to instantiate a session manager proxy.

Example 64: *Instantiating a Session Manager Proxy*

```
// C++
SessionManagerClient session_mgr;
SessionManagerClient* session_mgr_ptr = &session_mgr;
```

Start a session

After instantiating a session manager proxy, a client can then start a session for the desired service's group using the session manager's

`begin_session()` method. `begin_session()` has the following signature:

```
// C++
virtual void
begin_session(
    const IT_Bus::String &endpoint_group,
    const IT_Bus::ULong preferred_renew_timeout,
    SessionInfo &session_info
) IT_THROW_DECL(IT_Bus::Exception) = 0;
```

The `begin_session()` function takes the following input parameters:

- `endpoint_group`—the endpoint group name, which corresponds to the default group name set in the server's configuration scope as described in [“Configuring the server” on page 169](#).
- `preferred_renew_timeout`—the preferred session duration in seconds. If the specified duration is less than the value specified by the session manager's `min_session_timeout` configuration setting, it will be set to the configured minimum value. If the specified duration is higher than the value specified by the session manager's `max_session_timeout` configuration setting, it will be set to the configured max value.

And the following output parameter:

- `session_info`—a sequence complex type that contains the session id, `session_id`, and the actual assigned session duration, `renew_timeout`.

Example 65 shows the client code to begin a session for `SM_Demo`.

Example 65: *Beginning a Session*

```
// C++
...
IT_Bus_Services::IT_SessionManager::SessionId group_session;

int
main(int argc, char* argv[])
{
    ...
    // Begin a session
    session_mgr.begin_session("SM_Demo", 20, session_info);
    cout << "Begin session invoked" << endl;

    // Retrieve the session ID from the response
    group_session = session_info.getsession_id();
    cout << "Got session!" << endl << endl;
    ...
}
```

Get a list of endpoints in the group

The session manager hands out sessions for a group of services, so in order to get an individual service upon which to make requests a client needs to get a list of the services in the session's group. The session manager proxy's `get_all_endpoints()` method returns a list of all endpoints registered to the specified group. `get_all_endpoints()` has the following signature:

```
// C++
virtual void
get_all_endpoints(
    const SessionId &session_id,
    EndpointList &endpoints
) IT_THROW_DECL(IT_Bus::Exception) = 0;
```

The `get_all_endpoints()` function takes the following input parameter:

- `session_id`—the session ID for which you are requesting services (obtained in the previous step).

And the following output parameter:

- `endpoints`—the list of services available. If the group has no services, the list will be empty.

[Example 66](#) shows how to get the list of services for a group.

Example 66: *Retrieving the List of Services in a Group*

```
//C++
// Get the endpoints for the session.
IT_Bus_Services::IT_SessionManager::EndpointList endpoint_list;

// Must provide the session ID
// Without a valid session ID, the session manager will refuse
// the request
session_mgr.get_all_endpoints(
    group_session,
    endpoint_list
);
```

Create a proxy for the requested service

The client can use any of the services returned by `get_all_endpoints()` to instantiate a service proxy.

Because the session manager simply returns the services in the order the services registered with the session manager, the clients are responsible for circulating through the list or else they will all make requests on only one service in the group. Also, because the session manager does not force all members of a group to implement the same interface, you might need to have your clients check each service to see if it implements the correct interface by checking the reference's service name as shown in [Example 67](#).

Example 67: *Checking the Service Reference for its Interface*

```
//C++
IT_Bus::Reference& endpoint = endpoint_list[0];
if (endpoint.get_service_name() ==
    QName("", "SOAPService",
    "http://www.ionac.com/session_management")
)
{
    // Instantiate a SOAPService proxy
}
else
{
    // do something else
}
```

[Example 68](#) shows the client code for creating a `GreeterClient` proxy from an endpoint reference.

Example 68: *Instantiate a Proxy Server*

```
// C++
GreeterClient client(endpoint_list[0]);
```

Create a session header

Services that are being managed by the session manager will only accept requests that include a valid session header. [Example 69](#) shows how to send the session ID in a header by initializing the `sessionIDContext` header context. For more details about the context API used in this example, see [“Artix Contexts” on page 179](#).

Example 69: *Initialize the sessionIDContext Header Context*

```
// C++
using namespace session_management;
using namespace IT_Bus;
using namespace IT_Bus_Services::IT_SessionManager;
...
const QName DEMO_SESSION_ID_CONTEXT_NAME(
    "",
    "sessionIDContext",
    "http://ws.iona.com/sessionmanager"
);
...
// The session name and session group must be added to each
// request Without valid entries, the session endpoint manager
// will reject the request
ContextRegistry* registry = bus->get_context_registry();
ContextCurrent& current = registry->get_current();
ContextContainer* request_contexts = current.request_contexts();

AnyType* attr = request_contexts->get_context(
    DEMO_SESSION_ID_CONTEXT_NAME,
    true
);

if (0 == attr)
{
    cerr << endl << "Error : Unable to access Session Context"
        << endl;
    return -1;
}
```


Example 69: *Initialize the sessionIDContext Header Context*

```

}

SessionId* session_attr = dynamic_cast<SessionId*> (attr);

if (0 == session_attr)
{
    cerr << endl << "Error : Unable to cast Session Context"
        << endl;
    return -1;
}
session_attr->setname(group_session.getname());
session_attr->setendpoint_group(
    group_session.getendpoint_group()
);

```

Make requests on service proxy

Once the session information is added to the proxy's port information, the client can invoke operations on the endpoint as it would a non-managed service. If the endpoint rejects the request because the client's session is not valid, an exception is raised.

Renewing a session

If a client is going to use a session for a longer than the duration the session was granted, the client will need to renew its session or the session will timeout. A session is renewed using the session manager proxy's `renew_session()` method. `renew_session()` has the following signature:

```

// C++
virtual void
renew_session(
    const SessionInfo &session_info,
    IT_Bus::ULong &renew_timeout
) IT_THROW_DECL((IT_Bus::Exception)) = 0;

```

The `renew_session()` function takes the following input parameter:

- `session_info`—a sequence complex type that contains the session id, `session_id`, and the preferred session duration, `renew_timeout`.

And the following output parameter

- `renew_timeout`—the actual assigned session duration, in seconds.

If the renewal is unsuccessful, an

`IT_Bus_Services::renewSessionFaultException` is raised.

End the session

When a client is finished with a session managed service, it should explicitly end its session. This will ensure that the session will be freed up immediately. A session is ended using the session manager proxy's `end_session()` method. `end_session()` has the following signature:

```
// C++
virtual void
end_session(
    const SessionId &session_id
) IT_THROW_DECL((IT_Bus::Exception)) = 0;
```

[Example 70](#) shows how to end a session.

Example 70: Ending a Session

```
//C++
cout << "Ending session" << endl;
session_mgr.end_session(group_session);
```

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.

In this chapter

This chapter discusses the following topics:

Introduction to Contexts	page 180
Pre-Registered Contexts	page 192
Reading and Writing Context Data	page 196
Configuration Context Example	page 209
Header Context Example	page 220
Header Contexts in Three-Tier Systems	page 240

Introduction to Contexts

Overview

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.

In this section

This section contains the following subsections:

Configuration Contexts	page 181
Header Contexts	page 184
Reading and Writing Basic Types	page 201
Registering Contexts	page 186

Configuration Contexts

Overview

Artix *configuration contexts* provide a general purpose mechanism for configuring Artix transports, bindings and interceptors. Contexts enable you to configure both client-side settings (request contexts) and server-side settings (reply contexts).

Currently, configuration contexts are used mainly to configure WSDL port extensors (transport configuration). For example, [Figure 17](#) gives an overview of the context mechanism for configuring WSDL port extensors.

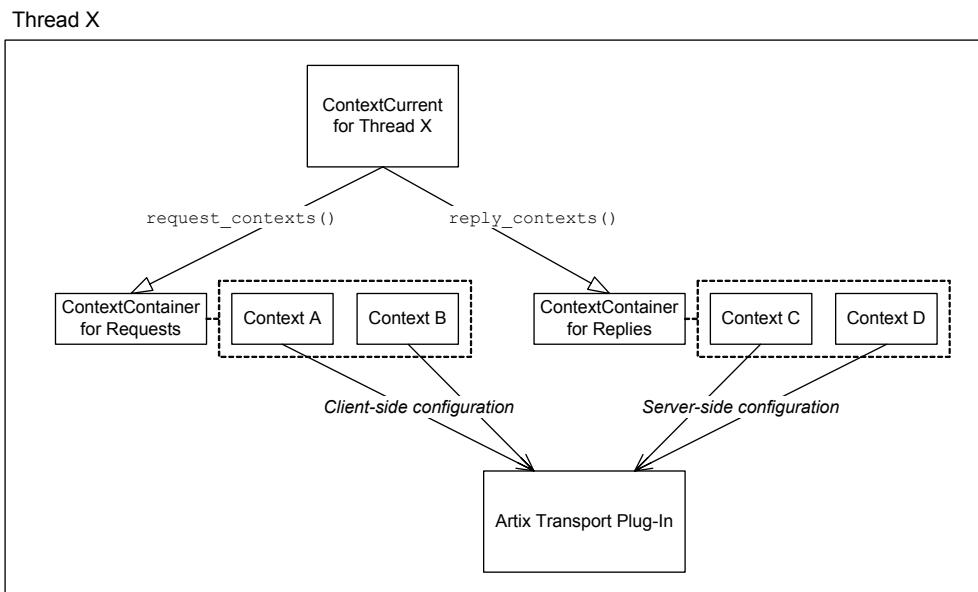


Figure 17: Overview of Configuration Contexts

Thread affinity

Context data is held in thread-specific storage, so that different threads can have different configurations. The root object for obtaining thread-specific data is the `IT_Bus::ContextCurrent` object.

Request contexts

Request contexts are used to read or modify the client-side properties of transports, bindings or interceptors.

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 request contexts.

Reply contexts

Reply contexts are used to read or modify the server-side properties of transports, bindings or interceptors.

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 reply contexts.

Schema-based API

In general, Artix lets you configure a WSDL port either by setting the port extensor attributes in the WSDL contract or by programming. The Artix configuration context mechanism unifies the two approaches by basing both the WSDL port extensors and the programming API on an XML schema, as follows:

- *WSDL contract*—the schema defines WSDL port extensor elements that can be initialized in the WSDL contract.
- *By programming*—the schema types are mapped to C++ using the WSDL-to-C++ compiler and then used as context data types.

Note: For convenience, Artix pre-compiles the schemas to C++ and makes the resulting stub code available in a library.

Schemas for configuration contexts

The following Artix schemas define data types that are used as configuration contexts:

- `http-conf.xsd`—defines the `<http-conf:client>` and `<http-conf:server>` WSDL port extensors that configure the HTTP transport.
- `mq.xsd`—defines the `<mq:client>` and `<mq:server>` WSDL port extensors that configure the MQ-Series transport.
- `i18n-context.xsd`—defines the `<i18n-context:client>` and `<i18n-context:server>` WSDL extensors that configure internationalization.

- `bus-security.xsd`—defines the `<bus-security:security>` WSDL port extensor that configure Artix security.
 - `corba.xsd`—enables you to define the CORBA Principal value programmatically.
-

Configuration context API

The following appendices provide more details on the configuration context programming interface:

- [“http-conf Context Data Types” on page 575](#)
- [“MQ-Series Context Data Types” on page 585](#)

Header Contexts

Overview

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 will be embedded in a SOAP header, as shown in [Figure 18](#).

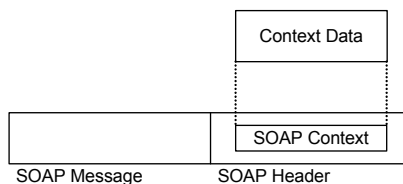


Figure 18: *Inserting Context Data into a SOAP Header*

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 will be embedded within a CORBA header as a GIOP service context—see [Figure 19](#).

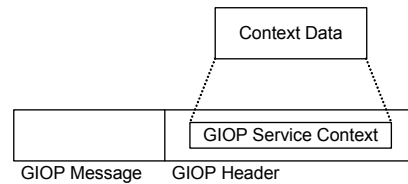


Figure 19: *Inserting Context Data into a GIOP Service Context*

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

Overview

Figure 20 shows an overview of what happens when you register a context data type with the context registry object.

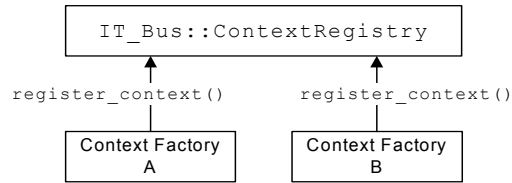


Figure 20: Registering Context Types with a Context Container

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 (for example, `ContextSchema_wsdlTypeFactory.cxx`), which creates static instances of the relevant type factories.

Getting a context registry instance

To get a reference to a context registry instance, you call the `IT_Bus::get_context_registry()` function, shown in [Example 71](#).

Example 71: The `IT_Bus::get_context_registry()` Function

```

// C++
namespace IT_Bus {
class IT_BUS_API Bus
{
public:
virtual ContextRegistry*
get_context_registry() = 0;
...
}
}

```

Example 71: *The `IT_Bus::get_context_registry()` Function*

```
};
};
```

Registering a configuration context

In practice, you do not need to register a configuration context unless you are implementing your own Artix plug-in. All of the standard Artix configuration contexts are pre-registered (see [“Pre-Registered Contexts” on page 192](#)).

You can register the following kinds of configuration context:

- [Registering a serializable configuration context](#)
- [Registering a non-serializable configuration context](#)

Registering a serializable configuration context

A serializable configuration context is a data type that inherits from the `IT_Bus::AnyType` base class. [Example 72](#) shows the signature of the `register_context()` function in the `IT_Bus::ContextRegistry` class, which is used to register a serializable configuration context.

Example 72: *The `register_context()` Function for Serializable Configuration 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 configuration contexts are given in [“Pre-Registered Contexts” on page 192](#).
- `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 configuration contexts, this flag should *not* be supplied (defaults to `false`).

Registering a non-serializable configuration context

A non-serializable configuration context can be any C++ type (that is, not necessarily inheriting from `IT_Bus::AnyType`). [Example 73](#) shows the signature of the `register_context_data()` function in the `IT_Bus::ContextRegistry` class, which is used to register a non-serializable configuration context.

Example 73: The `register_context_data()` Function for Non-Serializable Configuration 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 context 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 74](#) shows the signature of the `register_context()` function in the `IT_Bus::ContextRegistry` class, which is used to register a header context data type for the SOAP protocol.

Example 74: *The register_context() Function for SOAP Contexts*

```
// C++
namespace IT_Bus {
    class IT_BUS_API ContextRegistry
    {
    public:
        virtual Boolean
        register_context(
            const QName& context_name,
            const QName& context_type,
            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.

Registering a CORBA header context

Example 75 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 75: *The register_context() Function for CORBA Contexts*

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

Pre-Registered Contexts

Overview

This section provides a list of names and header files for the pre-registered configuration contexts.

Schema directory

The schemas for the Artix configuration contexts are located in the following directory:

```
ArtixInstallDir/artix/Version/schemas
```

Header files

The header files for the Artix configuration contexts are located in the following directory:

```
ArtixInstallDir/artix/Version/include/it_bus_pdk/context_attrs
```

Library

To gain access to the context stubs, you should link with the following library:

Windows

```
ArtixInstallDir/artix/Version/lib/it_context_attribute.lib
```

UNIX

```
ArtixInstallDir/artix/Version/lib/it_context_attribute.so
```

```
ArtixInstallDir/artix/Version/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.

```
IT_ContextAttributes::HTTP_CLIENT_OUTGOING_CONTEXTS
```

This context enables you to specify HTTP context data for inclusion with the next outgoing client request.

For this QName, `get_context()` returns the following data type:

```
IT_ContextAttributes::clientType
```


To use this context, include the following header file:

```
it_bus_pdk/context_attrs/http_conf_xsdTypes.h
```

`IT_ContextAttributes::HTTP_CLIENT_INCOMING_CONTEXTS`

This context enables you to read context data received with the last HTTP reply on the client side.

For this QName, `get_context()` returns the following data type:

```
IT_ContextAttributes::clientType
```

To use this context, include the following header file:

```
it_bus_pdk/context_attrs/http_conf_xsdTypes.h
```

`IT_ContextAttributes::HTTP_SERVER_OUTGOING_CONTEXTS`

This context enables you to specify HTTP context data for inclusion with the server's reply.

For this QName, `get_context()` returns the following data type:

```
IT_ContextAttributes::serverType
```

To use this context, include the following header file:

```
it_bus_pdk/context_attrs/http_conf_xsdTypes.h
```

`IT_ContextAttributes::HTTP_SERVER_INCOMING_CONTEXTS`

This context enables you to read context data received with the current HTTP request on the server side.

For this QName, `get_context()` returns the following data type:

```
IT_ContextAttributes::serverType
```

To use this context, include the following header file:

```
it_bus_pdk/context_attrs/http_conf_xsdTypes.h
```

`IT_ContextAttributes::CORBA_CONTEXT_ATTRIBUTES`

This context can be used to access and modify the CORBA Principal.

For this QName, `get_context()` returns the following data type:

```
IT_ContextAttributes::CORBAAttributesType
```

To use this context, include the following header file:

```
it_bus_pdk/context_attrs/corba_xsdTypes.h
```

`IT_ContextAttributes::SECURITY_SERVER_CONTEXT`

This context can be used to modify Bus security settings both on the client side and on the server side.

For this QName, `get_context()` returns the following data type:

`IT_ContextAttributes::BusSecurity`

To use this context, include the following header file:

`it_bus_pdk/context_attrs/bus_security_xsdTypes.h`

`IT_ContextAttributes::MQ_CONNECTION_ATTRIBUTES`

For this QName, `get_context()` returns the following data type:

`IT_ContextAttributes::MQConnectionAttributesType`

To use this context, include the following header file:

`it_bus_pdk/context_attrs/mq_xsdTypes.h`

`IT_ContextAttributes::MQ_OUTGOING_MESSAGE_ATTRIBUTES`

For this QName, `get_context()` returns the following data type:

`IT_ContextAttributes::MQMessageAttributesType`

To use this context, include the following header file:

`it_bus_pdk/context_attrs/mq_xsdTypes.h`

`IT_ContextAttributes::MQ_INCOMING_MESSAGE_ATTRIBUTES`

For this QName, `get_context()` returns the following data type:

`IT_ContextAttributes::MQMessageAttributesType`

To use this context, include the following header file:

`it_bus_pdk/context_attrs/mq_xsdTypes.h`

`IT_ContextAttributes::PRINCIPAL_CONTEXT_ATTRIBUTE`

Calling `get_context()` returns the Principal as an

`IT_Bus::StringHolder` instance.

No header file is required to use this context.

`IT_ContextAttributes::I18N_INTERCEPTOR_SERVER_QNAME`

For this QName, `get_context()` returns the following data type:

`IT_ContextAttributes::ServerConfiguration`

To use this context, include the following header file:

`it_bus_pdk/context_attrs/i18n_context_xsdTypes.h`

`IT_ContextAttributes::I18N_INTERCEPTOR_CLIENT_QNAME`

For this QName, `get_context()` returns the following data type:

`IT_ContextAttributes::ClientConfiguration`

To use this context, include the following header file:

`it_bus_pdk/context_attrs/i18n_context_xsdTypes.h`

`IT_ContextAttributes::HTTP_ENDPOINT_URL`

Calling `get_context()` returns the HTTP endpoint URL as an

`IT_Bus::StringHolder` instance.

No header file is required to use this context.

`IT_ContextAttributes::SERVER_OPERATION_CONTEXT`

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.

For this QName, `get_context()` returns the following data type:

`IT_Bus::ServerOperationContext`

To use this context, include the following header files:

```
it_bus_pdk/context_attrs/context_types.h
it_bus/operation.h
```

Reading and Writing Context Data

Overview

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.

In this section

This section contains the following subsections:

Getting a Context Instance	page 197
Reading and Writing Basic Types	page 201
Reading and Writing User-Defined Types	page 203
Reading and Writing Custom Types	page 205
Durability of Context Settings	page 208

Getting a Context Instance

Overview

Figure 21 shows an overview of how context data instances are accessed for writing and reading in an Artix application.

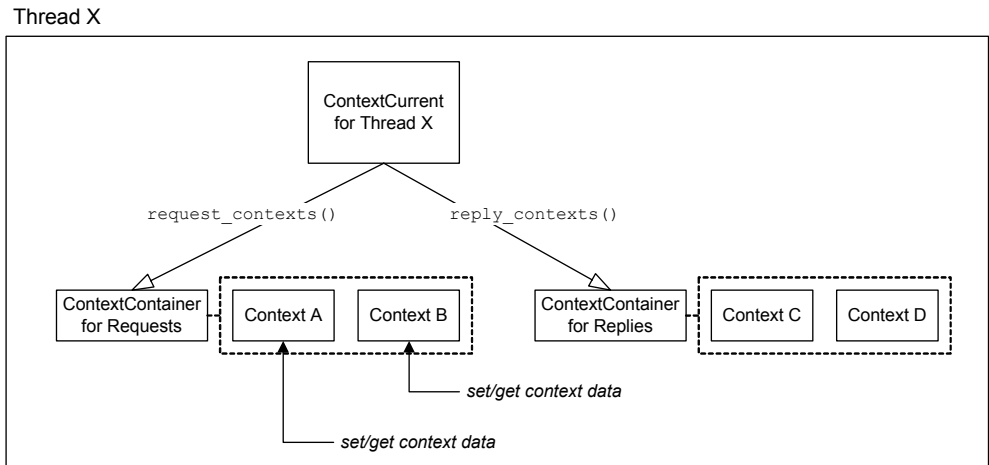


Figure 21: Overview of Context Data in a Multi-Threaded Application

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 76](#).

Example 76: 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 77](#) 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 77: 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;
    };
}
```

Context containers

A *context container* is an object that holds a collection of contexts associated with a particular thread. There are two kinds of context container:

- *Request context container*—contains the following kinds of context:
 - ◆ Configuration contexts for the client-side settings,
 - ◆ Header contexts to send in outgoing request messages,
 - ◆ Header contexts received from incoming request messages.
- *Reply context container*—contains the following kinds of contexts:
 - ◆ Configuration contexts for the server-side settings,
 - ◆ Header contexts to send in outgoing reply messages,
 - ◆ Header contexts received from incoming reply messages.

ContextContainer class

[Example 78](#) shows the declaration of the `IT_Bus::ContextContainer` class, which defines member functions for getting and setting context objects.

Example 78: 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
        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;
    };
};
```

Example 78: *The `IT_Bus::ContextContainer` Class*

```

// 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

Overview

To insert and extract a basic type, *BasicType*, you must use its corresponding *BasicTypeHolder* 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 285](#).

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.ionac.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

Overview

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 79](#).

Example 79: 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">
    ...
  </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

Overview

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 80: 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 81](#) 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 81: 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);

// ServerOperation is wrapped in a template ContextT class.
ServerOperation& server_op = operation->get_data();
```

Durability of Context Settings

Overview

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 to 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 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 217](#).

Configuration Context Example

Overview

This section shows how to modify the settings in a configuration 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.

In this section

This section contains the following subsections:

HTTP-Conf Schema	page 210
Setting a Configuration Context on the Client Side	page 214
Setting a Configuration Context on the Server Side	page 217

HTTP-Conf Schema

Overview

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/artix/Version/schemas/http-conf.xsd
```

http-conf:clientType XML definition

[Example 82](#) gives an extract from the `http-conf` schema, showing part of the definition of the `http-conf:clientType` complex type.

Example 82: Definition of the `http-conf:clientType` Type

```
<xs:schema
  targetNamespace="http://schemas.iona.com/transport/http/conf
  igation"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:http-conf="http://schemas.iona.com/transport/http/conf
  igation"
  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"
```

Example 82: *Definition of the http-conf:clientType Type*

```

...
                                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 83](#). 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 83: *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);

```

Example 83: C++ Mapping of `http-conf:clientType` Type

```

        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 84](#). 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 84: 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/artix/Version/lib/it_context_attribute.lib
```

UNIX

```
ArtixInstallDir/artix/Version/lib/it_context_attribute.so
```

```
ArtixInstallDir/artix/Version/lib/it_context_attribute.sl
```

```
ArtixInstallDir/artix/Version/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 Configuration Context on the Client Side

Overview

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 85](#) shows sample code from a client main function, which shows how to initialize `http-conf:clientType` context data in the current thread.

Example 85: Client 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         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 =
```

Example 85: *Client Main Function Setting a Configuration Context*

```

        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);

        // Modify the Send/Receive timeouts
8      http_client_config->setSendTimeout(2000);
        http_client_config->setReceiveTimeout(600000);
        ...
    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Error : Unexpected error occurred!"
            << 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 all 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

Overview

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 86](#) shows sample code from a server main function, which shows how to initialize `http-conf:serverType` configuration context data.

Example 86: *Client 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();
```

Example 86: *Client Main Function Setting a Configuration Context*

```

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");
    ...
    GreeterImpl servant(bus);
    bus->register_servant(
        servant,
        "../etc/hello_world.wsdl",
        service_name
    );
}
catch(IT_Bus::Exception& e)
{
    cout << endl << "Error : Unexpected error occurred!"
        << 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 `demos/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).

Note: Currently, the `ContextContainer` object returned by `get_configuration_context()` can be used *only* to set an endpoint URL on the server side.

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.

Header Context Example

Overview

This section provides a detailed discussion of the custom SOAP header demonstration, which shows you how to propagate context data in a SOAP header.

In this section

This section contains the following subsections:

Custom SOAP Header Demonstration	page 221
SOAP Header Context Schema	page 223
Declaring the SOAP Header Explicitly	page 226
Client Main Function	page 229
Server Main Function	page 234
Service Implementation	page 237

Custom SOAP Header Demonstration

Overview

The examples in this section are based on the custom SOAP header demonstration, which is located in the following Artix directory:

ArtixInstallDir/artix/Version/demos/advanced/custom_soap_header

Figure 22 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.

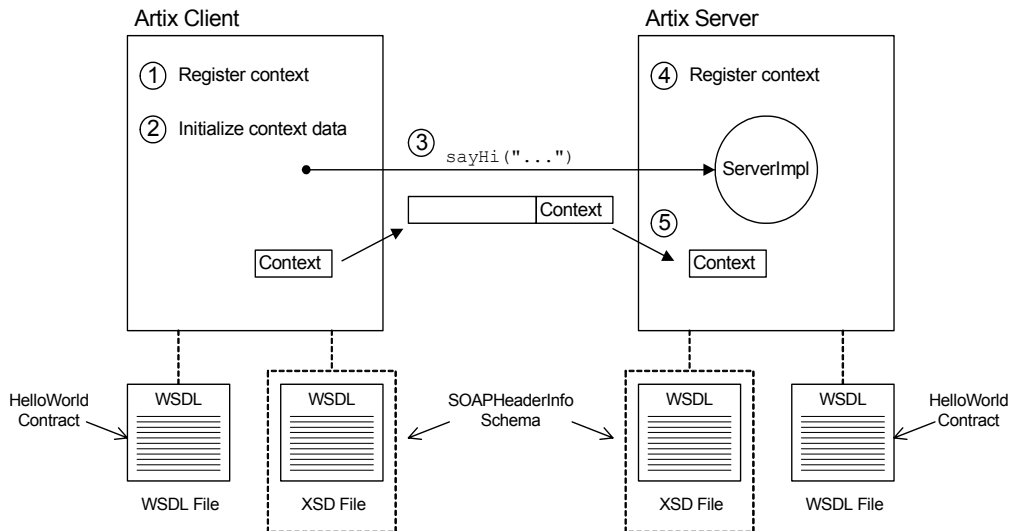


Figure 22: Overview of the Custom SOAP Header Demonstration

Transmission of context data

As illustrated in [Figure 22](#), 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 `demos/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

Overview

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 87](#) 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 87](#) is a pure schema declaration, *not* a WSDL declaration.

Example 87: 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 87](#). This command generates the following C++ stub files:

```
contextTypes_xsdTypes.h
contextTypes_xsdTypesFactory.h
contextTypes_xsdTypes.cxx
contextTypes_xsdTypesFactory.cxx
```

SOAPHeaderInfo C++ mapping

[Example 88](#) shows how the schema from [Example 87 on page 223](#) maps to C++, to give the `soap_interceptor::SOAPHeaderInfo` C++ class.

Example 88: 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);
        ...
    };
};
```


Example 88: *C++ Mapping of the SOAPHeaderInfo Context Type*

```
};  
...  
}
```

Declaring the SOAP Header Explicitly

Overview

There are two different approaches you can take with SOAP headers:

- *Implicit SOAP header*—(the approach taken in [Example 87 on page 223](#)) 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/artix/Version/demos/advanced/soap_header_binding`

SOAP header declaration

[Example 89](#) shows how to declare a SOAP header, of `SOAPHeaderData` type, explicitly in a WSDL contract.

Example 89: *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/transport/http/configuration"
  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">
```

Example 89: SOAP Header Declared in the WSDL Contract

```

<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>
    </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"/>
    </input>
    4 <soap:header message="tns:header_message"
      part="header_info"
      use="literal"/>
  </operation>
</binding>

```

Example 89: SOAP Header Declared in the WSDL Contract

```

        </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 87 on page 223](#)). 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

Overview

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 SOAP container*—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 SOAP current object*—before invoking any operations, the client obtains an instance of the header context data from a SOAP current object. After initializing the header context data, any operations invoked from the current thread will include the header context data.

Client main function

[Example 90](#) 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 90: *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"
```

Example 90: *Client Main Function Setting a SOAP Context*

```

#include "GreeterClient.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);
        GreeterClient client;

3         ContextRegistry* context_registry =
            bus->get_context_registry();

4         // Create QName objects needed to define a context
        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");

8         // Register the context with the ContextRegistry
        context_registry->register_context(
            principal_ctx_name,
            principal_ctx_type,
            principal_message_name,
            principal_part_name
        );
    }
}

```

Example 90: *Client Main Function Setting a SOAP Context*

```

9      // Obtain a reference to the ContextCurrent
ContextCurrent& context_current =
    context_registry->get_current();

10     // Obtain a pointer to the RequestContextContainer
ContextContainer* context_container =
    context_current.request_contexts();

11     // Obtain a reference to the context
AnyType* info = context_container->get_context(
    principal_ctx_name,
    true
);

12     // Cast the context into a SOAPHeaderInfo object
SOAPHeaderInfo* header_info =
    dynamic_cast<SOAPHeaderInfo*> (info);

    // Create the content to be added to the header
const String originator("IONA Technologies");
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 occurred!"
    << 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 87 on page 223](#)).
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 87 on page 223](#).
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

Overview

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 92 on page 237](#)).

Server main function

[Example 91](#) shows sample code from the server main function, which registers the `SOAPHeaderInfo` context type and then creates and registers a `GreeterImpl` servant object.

Example 91: Server Main Function Registering a SOAP 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 "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();
```

Example 91: *Server Main Function Registering a SOAP Context*

```

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
        );

      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 87 on page 223](#).
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

Overview

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 92](#) 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 92: *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();

2   // Create QName objects needed to define a context
    const QName principal_ctx_name(
        "",
        "SOAPHeaderInfo",
        ""
    );
};
```

Example 92: *sayHi Operation Accessing a SOAP Context*

```

3 // Obtain a reference to the ContextCurrent
ContextCurrent& context_current =
    context_registry->get_current();

4 // Obtain a pointer to the RequestContextContainer
ContextContainer* context_container =
    context_current.request_contexts();

5 // Obtain a reference to the context
AnyType* info = context_container->get_context(
    principal_ctx_name
);

6 // Cast the context into a SOAPHeaderInfo object
SOAPHeaderInfo* header_info =
    dynamic_cast<SOAPHeaderInfo*> (info);

7 // Extract the application specific SOAP header information
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()`.

Header Contexts in Three-Tier Systems

Overview

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 23 shows an example of a three-tier system where a request context is propagated automatically from tier to tier.

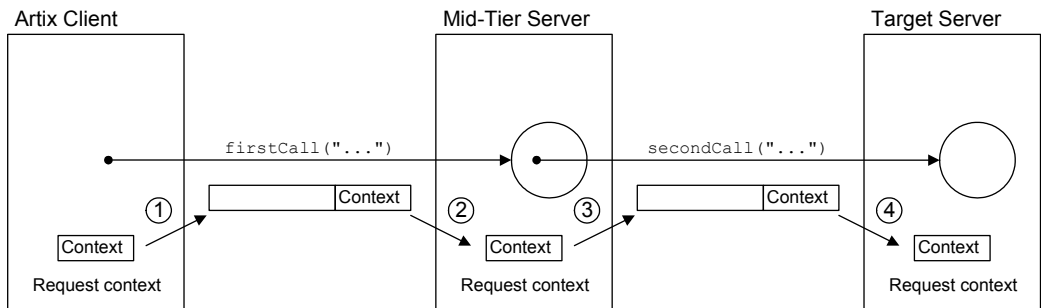


Figure 23: Propagation of a Request Context in a Three-Tier System

Context propagation steps

In [Figure 23](#), 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.

Artix Data Types

This chapter presents the XML schema data types supported by Artix and describes how these data types map to C++.

In this chapter

This chapter discusses the following topics:

Including and Importing Schema Definitions	page 244
Simple Types	page 246
Complex Types	page 288
Wildcarding Types	page 333
Occurrence Constraints	page 349
Nillable Types	page 368
Substitution Groups	page 390
SOAP Arrays	page 400
IT_Vector Template Class	page 412
Unsupported XML Schema Constructs in Artix	page 419

Including and Importing Schema Definitions

Overview

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 93](#) shows an example of an XML schema that includes another XML schema.

Example 93: *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 94](#) shows the contents of the included schema file, `included.xsd`.

Example 94: *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

Overview

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.

In this section

This section contains the following subsections:

Atomic Types	page 247
String Type	page 249
NormalizedString and Token Types	page 254
QName Type	page 259
Date and Time Types	page 261
Decimal Type	page 263
Integer Types	page 265
Binary Types	page 268
Deriving Simple Types by Restriction	page 275
List Type	page 278
Union Type	page 280
Holder Types	page 285
Unsupported Simple Types	page 287

Atomic Types

Overview

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 2: *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
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

Table 2: Simple Schema Type to Simple Bus Type Mapping

Schema Type	Bus Type
xsd:Name	IT_Bus::Name
xsd:NCName	IT_Bus::NCName
xsd:ID	IT_Bus::ID
xsd:QName	IT_Bus::QName <i>(SOAP only)</i>
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

Overview

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`.

C++ 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 *Deploying and Managing 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 3](#) shows some of commonly used IANA character set names:

Table 3: *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 95](#) shows three different constructors that can be used to initialize an `IT_UString` with a narrow character string.

Example 95: *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 96](#) shows the constructor that can be used to initialize an `IT_UString` with an array of 16-bit characters (represented by unsigned `short*`).

Example 96: 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 97](#) shows the constructor that can be used to initialize an `IT_UString` with an array of `wchar_t` characters.

Example 97: 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 98](#) 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 98: 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 99](#) 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 99: 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());

// 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 *Deploying and Managing Artix Solutions* document.

NormalizedString and Token Types

Overview

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 4](#).

Table 4: Description of token and Types Derived from token

XML Schema Type	Sample Value	Description of Value
<code>xsd:token</code>	Only single spaces; no leading or trailing!	Like an <code>xsd:normalizedString</code> type, except that there can be no sequences of two or more spaces (0x20) and no leading or trailing spaces.
<code>xsd:language</code>	en-US	Any language identification tag as specified in RFC 3066 (http://www.ietf.org/rfc/rfc3066.txt).
<code>xsd:NMTOKEN</code>	NoSpacesAllowed	Like an <code>xsd:token</code> type, except that spaces (0x20) are disallowed (see “ Formal definitions ” on page 255).
<code>xsd:NMTOKENS</code>	Tok01 Tok02 Tok03	A list of <code>xsd:NMTOKEN</code> items, using the space character as a delimiter.
<code>xsd:Name</code>	RestrictFirstChar	Like an <code>xsd:token</code> type, except that the first character is restricted to be one of Letter, ‘_’, or ‘:’ (see “ Formal definitions ” on page 255).
<code>xsd:NCName</code>	NoColonsAllowed	Like an <code>xsd:Name</code> type, except that colons, ‘:’, are disallowed (a <i>non-colonized name</i>). See “ Formal definitions ” on page 255. This type is useful for constructing identifiers that use the colon, ‘:’, as a delimiter. For example, the <code>NCName</code> type is used both for the prefix and the local part of an <code>xsd:QName</code> .

Table 4: Description of token and Types Derived from token

XML Schema Type	Sample Value	Description of Value
xsd:ID	LikeNCName	Like an <code>xsd:NCName</code> type. The <code>xsd:ID</code> 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 2 on page 247](#). 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 5](#).

Table 5: *Validity Testing Functions for Normalized Strings and Tokens*

XML Schema Type	Validity Testing Function
<code>xsd:normalizedString</code>	<pre>static bool IT_Bus::NormalizedString::is_valid_normalized_string(const String& value)</pre>
<code>xsd:token</code>	<pre>static bool IT_Bus::Token::is_valid_token(const String& value)</pre>
<code>xsd:language</code>	<pre>static bool IT_Bus::Language::is_valid_language(const String& value)</pre>
<code>xsd:NMTOKEN</code>	<pre>static bool IT_Bus::NMToken::is_valid_nmtoken(const String& value)</pre>
<code>xsd:Name</code>	<pre>static bool IT_Bus::Name::is_valid_name(const String& value)</pre>
<code>xsd:NCName</code>	<pre>static bool IT_Bus::NCName::is_valid_ncname(const String& value)</pre>
<code>xsd:ID</code>	<pre>static bool IT_Bus::ID::is_valid_id(const String& value)</pre>

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 412](#).

C++ 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

Overview

`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*.

Note: In Artix 1.2.1, the mapping from `xsd:QName` to `IT_Bus::QName` is supported only for the SOAP binding.

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

Overview

`xsd:dateTime` maps to `IT_Bus::DateTime`, which is declared in `<it_bus/date_time.h>`. `DateTime` has the following fields:

Table 6: *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)
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 7](#). Each of these types—for example, `xsd:time` and `xsd:day`—support a subset of the fields from `xsd:dateTime`. [Table 7](#) shows which fields are supported for each date and time type; the accessors for each field are given by [Table 6](#).

Table 7: *Member Fields Supported by Other Date and Time Types*

Date/Time Type	C++ Class	Supported Fields
<code>xsd:date</code>	<code>IT_Bus::Date</code>	year, month, day, local time zone flag, hour and minute offset from GMT.
<code>xsd:time</code>	<code>IT_Bus::Time</code>	hours, minutes, seconds, milliseconds, local time zone flag, hour and minute offset from GMT.
<code>xsd:gDay</code>	<code>IT_Bus::GDay</code>	day, local time zone flag, hour and minute offset from GMT.
<code>xsd:gMonth</code>	<code>IT_Bus::GMonth</code>	month, local time zone flag, hour and minute offset from GMT.
<code>xsd:gMonthDay</code>	<code>IT_Bus::GMonthDay</code>	month, day, local time zone flag, hour and minute offset from GMT.
<code>xsd:gYear</code>	<code>IT_Bus::GYear</code>	year, local time zone flag, hour and minute offset from GMT.
<code>xsd:gYearMonth</code>	<code>IT_Bus::GYearMonth</code>	year, month, local time zone flag, hour and minute offset from GMT.

Decimal Type

Overview

`xsd:decimal` maps to `IT_Bus::Decimal`, which is implemented by the IONA 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 8](#) for a list of supported operators.

Table 8: *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.

C++ 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

Overview

The XML schema defines the following unlimited precision integer types, as shown in [Table 9](#).

Table 9: *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::NonNegativeInteger`. 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 10](#) for a list of supported operators.

Table 10: *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;
```

C++ 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

Overview

There are two WSDL binary types, which map to C++ as shown in [Table 11](#):

Table 11: Schema to Bus Mapping for the Binary Types

Schema Type	Bus Type
xsd:base64Binary	IT_Bus::Base64Binary
xsd:hexBinary	IT_Bus::HexBinary

Encoding

The only 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).

IT_Bus::Base64Binary and IT_Bus::HexBinary classes

Both the `IT_Bus::Base64Binary` and the `IT_Bus::HexBinary` classes expose the following member functions to access the buffer value, as follows:

```
// 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::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 *borrow*s the buffer memory (in which case the binary buffer is not responsible for deleting it).

Allocating and deallocating binary buffers

[Example 100](#) shows the signatures of the binary buffer functions for allocating and deallocating binary buffers.

Example 100: 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 101 shows the signatures of the binary buffer functions for assigning and copying binary buffers.

Example 101: *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 102](#) shows the signatures of the binary buffer functions for accessing binary buffer data.

Example 102: 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 103](#) shows the signatures of the binary buffer functions for searching and comparing binary buffers.

Example 103: *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 104](#) shows the signatures of the binary buffer functions for concatenating binary buffers.

Example 104: *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).

C++ 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 105](#) shows how a server might implement the `echoHexBinary` operation.

Example 105: 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

Overview

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:

- `length`
 - `minLength`
 - `maxLength`
 - `pattern`
 - `enumeration`
 - `whiteSpace`
 - `maxInclusive`
 - `maxExclusive`
 - `minInclusive`
 - `minExclusive`
 - `totalDigits`
 - `fractionDigits`
-

Checked facets

The following facets are supported and checked at runtime:

- `enumeration`
-

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 106](#) 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 106: 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>
      ...
    </types>
  </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 107](#). The only values that can legally be set using the `set_value()` member function are the strings `RED`, `GREEN`, or `BLUE`.

Example 107: *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

Overview

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

Overview

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 `u1` 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 108](#) shows an example of a union type, `Union2`, which can contain either a `Union1` type or an enumerated string.

Example 108: 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"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:union>
</xsd:simpleType>
```

Example 108: *Definition of a Union Type in WSDL (Continued)*

```

        <enumeration value="Tweedledee"/>
    </xsd:restriction>
</xsd:simpleType>
</xsd:union>
</xsd:simpleType>

```

C++ mapping

The WSDL-to-C++ compiler maps the preceding WSDL ([Example 108 on page 281](#)) to the Union2 C++ class. An outline of this class is shown in [Example 109](#).

Example 109: *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()
    {

```

Example 109: *Mapping of Union2 to C++ (Continued)*

```

        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);

    Union2String &      getstring();
    const Union2String & getstring() const;
    void              setstring(const Union2String & val);
    // ...
};

```

The C++ mapping defines a pair of accessor and modifier functions, `getMemberType()` and `setMemberType()`, 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 `getAtomicType()` and `setAtomicType()` (for example, `getInt()` and `setInt()`).
- If the union member is a user-defined type, `UserType`, the functions are defined as `getUserType()` and `setUserType()`.
- 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 `getBaseType()` and `setBaseType()`.

C++ 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 110](#) shows how a client could use a proxy instance, `bc`, to invoke the `echoUnion` operation.

Example 110: *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) "
                << uOut.getstring().get_value().c_str() << endl;
            break;
    }
} catch (IT_Bus::FaultException &ex)
{
    // Handle exception (not shown) ...
}
```

Holder Types

Overview

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 *SimpleHolder* type derives from `IT_Bus::AnyType`. Accessor and modifier functions are used to insert and extract the *Simple* type from the *SimpleHolder* 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 12](#) shows the list of `Holder` types provided by Artix.

Table 12: *List of Artix Holder Types*

Built-In Type	Holder Type
<code>IT_Bus::Boolean</code>	<code>IT_Bus::BooleanHolder</code>
<code>IT_Bus::Byte</code>	<code>IT_Bus::ByteHolder</code>
<code>IT_Bus::Short</code>	<code>IT_Bus::ShortHolder</code>
<code>IT_Bus::Int</code>	<code>IT_Bus::IntHolder</code>
<code>IT_Bus::Long</code>	<code>IT_Bus::LongHolder</code>
<code>IT_Bus::String</code>	<code>IT_Bus::StringHolder</code>
<code>IT_Bus::Float</code>	<code>IT_Bus::FloatHolder</code>
<code>IT_Bus::Double</code>	<code>IT_Bus::DoubleHolder</code>
<code>IT_Bus::UByte</code>	<code>IT_Bus::UByteHolder</code>
<code>IT_Bus::UShort</code>	<code>IT_Bus::UShortHolder</code>
<code>IT_Bus::UInt</code>	<code>IT_Bus::UIntHolder</code>
<code>IT_Bus::ULong</code>	<code>IT_Bus::ULongHolder</code>
<code>IT_Bus::Decimal</code>	<code>IT_Bus::DecimalHolder</code>
<code>IT_Bus::QName</code>	<code>IT_Bus::QNameHolder</code>
<code>IT_Bus::DateTime</code>	<code>IT_Bus::DateTimeHolder</code>
<code>IT_Bus::HexBinary</code>	<code>IT_Bus::HexBinaryHolder</code>
<code>IT_Bus::Base64Binary</code>	<code>IT_Bus::Base64BinaryHolder</code>

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:duration  
xsd:ENTITY  
xsd:ENTITIES  
xsd:NOTATION  
xsd:IDREF  
xsd:IDREFS
```

Complex Types

Overview

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.

In this section

This section contains the following subsections:

Sequence Complex Types	page 289
Choice Complex Types	page 292
All Complex Types	page 296
Attributes	page 299
Attribute Groups	page 303
Nesting Complex Types	page 306
Deriving a Complex Type from a Simple Type	page 310
Deriving a Complex Type from a Complex Type	page 313
Arrays	page 323
Model Group Definitions	page 328

Sequence Complex Types

Overview

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 `PortTypeNameTypes.h` and `PortTypeNameTypes.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 355](#).

WSDL example

[Example 111](#) shows an example of a sequence, `SequenceType`, with three elements.

Example 111: *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 111) to the `SequenceType` C++ class. An outline of this class is shown in Example 112.

Example 112: 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, `getElementName()` and `setElementName()`.

C++ 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 113](#) shows how a client could use a proxy instance, `bc`, to invoke the `echoSequence` operation.

Example 113: 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

Overview

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 `PortTypeNameTypes.h` and `PortTypeNameTypes.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 359](#).

WSDL example

[Example 114](#) shows an example of a choice complex type, `ChoiceType`, with three elements.

Example 114: *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 114](#)) to the `SequenceType` C++ class. An outline of this class is shown in [Example 115](#).

Example 115: *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);

    ChoiceTypeDiscriminator get_discriminator() const
    {
        return m_discriminator;
    }

    IT_Bus::UInt get_discriminator_as_uint() const
    {
        return m_discriminator;
    }
}
```

Example 115: *Mapping of ChoiceType to C++*

```
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, *getElementName()* and *setElementName()*.

The member functions have the following effects:

- *setElementName()*—select the *ElementName* element, setting the discriminator to the *ElementName* label and initializing the value of *ElementName*.
- *getElementName()*—get the value of the *ElementName* element. You should always check the discriminator before calling the *getElementName()* accessor. If *ElementName* is not currently selected, the value returned by *getElementName()* is undefined.
- *get_discriminator()*—returns the value of the discriminator.

C++ 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 116](#) shows how a client could use a proxy instance, *bc*, to invoke the *echoChoice* operation.

Example 116: *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 {
```

Example 116: *Client Invoking an echoChoice Operation*

```
bc.echoChoice(cIn, cResult);

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

Overview

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 `PortTypeNameTypes.h` and `PortTypeNameTypes.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 117](#) shows an example of an all complex type, `AllType`, with three elements.

Example 117: *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 117](#)) to the `AllType` C++ class. An outline of this class is shown in [Example 118](#).

Example 118: *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, `getElementName()` and `setElementName()`.

C++ 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 119](#) shows how a client could use a proxy instance, `bc`, to invoke the `echoAll` operation.

Example 119: 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

Overview

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 `getAttribute()` 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 `getAttribute()` function returns a reference to the attribute value.

Optional attribute example

[Example 120](#) shows how to define a sequence type with a single optional attribute, `prop`, of `xsd:string` type (attributes are optional by default).

Example 120: 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 121](#) shows an outline of the C++ `SequenceType` class generated from [Example 120](#), which defines accessor and modifier functions for the optional `prop` attribute.

Example 121: Mapping an Optional Attribute to C++

```
// C++
class SequenceType : public IT_Bus::SequenceComplexType
{
public:
  SequenceType();
  ...
  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 122](#) shows how to define a sequence type with a single required attribute, `prop`, of `xsd:string` type.

Example 122: 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 123](#) shows an outline of the C++ `SequenceType` class generated from [Example 122 on page 301](#), which defines accessor and modifier functions for the required `prop` attribute.

Example 123: 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

Overview

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 124](#) 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 124: 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>
```

Example 124: *Example of Defining a Simple Attribute Group*

```

<complexType name="Package">
  <sequence> ... </sequence>
  <attributeGroup ref="tns:DimAttrGroup" />
</complexType>

</schema>

```

The preceding `Package` type defined in [Example 124 on page 303](#) is exactly equivalent to the `Package` type defined in [Example 125](#). In other words, referencing an attribute group has essentially the same effect as defining the attributes directly within the type.

Example 125: *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 126](#) shows how to define an attribute group, `DimAndColor`, which recursively references another attribute group, `DimAttrGroup`.

Example 126: *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">

```


Example 126: *Example of Defining a Nested Attribute Group*

```
<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 299](#).

Nesting Complex Types

Overview

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 127](#).

WSDL example

[Example 127](#) shows an example of a nested complex type, which features a choice complex type, `NestedChoiceType`, nested inside a sequence complex type, `SeqOfChoiceType`.

Example 127: 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="xsd: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 128](#) shows an outline of the generated C++ `NestedChoiceType` class.

Example 128: 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:
    ...
};
```

C++ mapping of SeqOfChoiceType

The XML schema sequence complex type, `SeqOfChoiceType`, has the `NestedChoiceType` nested inside it. [Example 129](#) shows an outline of the generated C++ `SeqOfChoiceType` class, which shows how the nested complex type is mapped within a sequence complex type.

Example 129: Mapping of `SeqOfChoiceType` to C++

```
// C++
class SeqOfChoiceType : public IT_Bus::SequenceComplexType
{
    ...
```

Example 129: *Mapping of SeqOfChoiceType to C++*

```

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.

C++ 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 119](#) shows how a client could use a proxy instance, `bc`, to invoke the `echoSeqOfChoice` operation.

Example 130: *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);
}

```

Example 130: *Client Invoking an echoSeqOfChoice Operation*

```
    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

Overview

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 131](#) 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 131: *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 132](#) 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 132: 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 133](#) shows an outline of the C++ `internationalPrice` class generated from [Example 132 on page 311](#).

Example 133: 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);
```

Example 133: *Mapping the internationalPrice Type to C++*

```
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

Overview

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 24](#) 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.

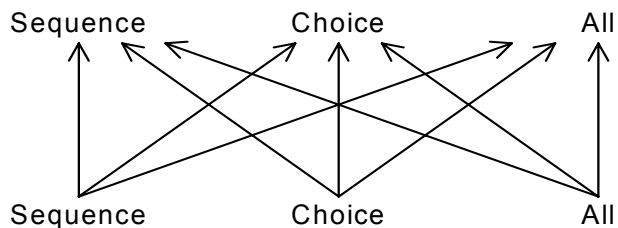


Figure 24: *Allowed Inheritance Relationships for Complex Types*

Derivation by restriction

Example 134 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 134: 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">
2   <complexContent>
3     <restriction base="tns:SimpleStruct">
4       <sequence>
          <element name="varFloat" type="float"/>
          <element name="varInt" type="int"/>
          <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 135 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 135: 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 135 on page 315](#), `SimpleStruct` and `DerivedStruct_BaseStruct`, map to C++ as shown in [Example 136](#).

Example 136: 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);
};
```

Example 136: C++ *Mapping of a Derived Sequence Type*

```

private:
    ...
};

typedef IT_AutoPtr<SimpleStruct> SimpleStructPtr;

...
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 136 on page 316](#) 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 `getElementName()` and `setElementName()` 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 137: *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 138 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 138: Calling `test_SimpleStruct()` with Derived Type Arguments

```

1 // C++
  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 139](#) 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 139: Calling `test_SimpleStruct()` with Base Type Arguments

```

1 // C++
  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 320](#)).

Arrays

Overview

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 140](#) shows how to define a one-dimensional string array, `ArrayOfString`, whose size can lie anywhere in the range 0 to unbounded.

Example 140: 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 141](#) shows how the `ArrayOfString` string array (from [Example 140 on page 324](#)) maps to C++.

Example 141: 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();
```

Example 141: *Mapping of ArrayOfString to C++*

```

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 355](#)). The accessor functions are superfluous, however, because the array’s elements are more easily accessed by invoking vector operations directly on the `ArrayOfString` class.

C++ example

[Example 142](#) 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 416](#)).

Example 142: *C++ 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 306](#) for a discussion of nested types). [Example 143](#) shows an example of how to define a two-dimensional string array, `ArrayOfArrayOfString`.

Example 143: *Definition of a Multi-Dimensional String Array*

```

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

```

Example 143: *Definition of a Multi-Dimensional String Array*

```

<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).

C++ example for multidimensional array

[Example 144](#) shows an example of how to allocate and initialize a multi-dimensional array, of `ArrayOfArrayOfString` type.

Example 144: *C++ 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 145](#) shows how an instance, `a2`, of `ArrayOfArrayOfString` type converts to an instance of `IT_Vector<ArrayOfString>` type by assignment.

Example 145: 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 412](#).
- The section on C++ ANSI vectors in *The C++ Programming Language*, third edition, by Bjarne Stroustrup.

Model Group Definitions

Overview

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 146](#) shows how to define a model group, `PassengerName`, which contains a sequence of elements.

Example 146: 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 147](#).

Example 147:*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 148](#) shows how to define a model group, `PassengerID`, which contains a choice of elements.

Example 148:*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 292](#)).

Recursive group references

[Example 149](#) shows how to define a model group, `Hop`, which recursively references another model group definition, `PassengerName`.

Example 149: 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 150](#).

Example 150: Hop Model Group Mapping to C++

```
// C++
class Hop : public IT_Bus::SequenceComplexType
{
public:
```

Example 150: *Hop Model Group Mapping to C++*

```

...
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);

private:
...
};

```

Repeated group references

[Example 151](#) shows how to define a model group, `TwoHops`, which references the `Hop` model group twice.

Example 151: *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 152](#).

Example 152: *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 $N+1^{\text{th}}$ repetition of a model group reference would generate a pair of functions, `getHop_N()` and `setHop_N()`.

Wildcarding Types

Overview

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:

- *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.

In this section

This section contains the following subsections:

anyURI Type	page 334
anyType Type	page 336
any Type	page 341

anyURI Type

Overview

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 153](#) shows the most important member functions from the `IT_Bus::AnyURI` class, which is the C++ mapping of `xsd:anyURI`.

Example 153: *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();
        ...
    };
};
```

Example 153:*The `IT_Bus::AnyURI` Class*

```
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 154](#) shows an example of a WSDL type, `DocReference`, that includes an attribute of `xsd:anyURI` type.

Example 154:*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>
```

C++ example

The following example code shows how to create an instance of the `DocReference` type defined in the preceding [Example 154](#). 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

Overview

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.cpp` 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 339](#).

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 111 on page 289](#)), 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 `xsd: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 `any`s are fully self-describing.

AnyHolder API

[Example 155](#) shows the public API from the `IT_Bus::AnyHolder` class, including all of the function for inserting and extracting data values.

Example 155:*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();
        IT_Bus::UByte & get_ubyte() ;
        IT_Bus::UShort & set_ushort();
        IT_Bus::UInt & get_uint();
        IT_Bus::ULong & set_ulong();
    };
};
```

Example 155:*The IT_Bus::AnyHolder Class*

```
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

Overview

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 anys with occurrence constraints, see [“Any Occurrence Constraints” on page 363](#).

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:

<code>##any</code>	(Default) Matches elements in any namespace, including unqualified elements.
<code>##local</code>	Matches an unqualified element (no namespace prefix appearing in the element name).
<code>##targetNamespace</code>	Matches elements in the current <code>targetNamespace</code> .

<code>##other</code>	Matches elements in any namespace apart from the current <code>targetNamespace</code> .
<code>Namespace</code>	Matches elements in the literal <code>Namespace</code> .
List of namespaces	A space-separated list of namespaces. The list can include literal namespaces, <code>##targetNamespace</code> , or <code>##local</code> .

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:

<code>strict</code>	(Default) A schema definition for the element type must be available and the element must conform to this definition.
<code>lax</code>	The parser checks only those parts of the element for which a schema definition is available.
<code>skip</code>	No checking is done against a schema; the element must simply be well-formed XML.

WSDL any example

[Example 156](#) 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 156: 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 156 on page 342](#) maps to the C++ `SequenceAny` class shown in [Example 157](#). The most important functions in `SequenceAny` are the `getany()` and `setany()` members, which access or modify the any element in the sequence.

Example 157:*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 158](#) shows the definition of a sample `foo` element, which can be inserted in place of an any element.

Example 158:*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">
```

Example 158: *Definition of fooType Type and foo Element*

```

<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>

```

C++ 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 159 shows the public API from the `IT_Bus::Any` class.

Example 159:*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(
```

Example 159:*The `IT_Bus::Any` Class*

```

        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
    );
    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
    );

```

Example 159:*The IT_Bus::Any Class*

```

// 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;

// 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

Overview

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.

In this section

This section contains the following subsections:

Element Occurrence Constraints	page 350
Sequence Occurrence Constraints	page 355
Choice Occurrence Constraints	page 359
Any Occurrence Constraints	page 363

Element Occurrence Constraints

Overview

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 323](#).

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 412](#)).

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 160](#) shows the definition of a sequence type, `SequenceType`, which contains a list of integer elements followed by a list of string elements.

Example 160:*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 161](#) shows an outline of the C++ `SequenceType` class generated from [Example 160 on page 352](#), which defines accessor and modifier functions for the `varInt` and `varString` elements.

Example 161:*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);
```


Example 161: *Mapping of SequenceType to C++*

```

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:
...
};

```

C++ 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 412.
- The section on C++ ANSI vectors in *The C++ Programming Language*, third edition, by Bjarne Stroustrup.

Sequence Occurrence Constraints

Overview

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 162](#) 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 162: 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 163](#) shows an outline of the C++ `CultureInfo` class generated from [Example 162 on page 355](#), which defines accessor and modifier functions for the `Name` and `Lcid` elements.

Example 163: 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;
    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);
};
```

Example 163: *Mapping CultureInfo to C++*

```
};
```

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 `setLcid()`—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.

C++ 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

Overview

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 164](#) shows the definition of a choice type, `ClubEvent`, with choice occurrence constraints. The choice type overall can be repeated 0 to unbounded times.

Example 164: 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 165](#) shows an outline of the C++ `ClubEvent` class generated from [Example 164 on page 359](#), which defines accessor and modifier functions for the `MemberName` and `GuestName` elements.

Example 165: 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 { ... }

    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;
```


Example 165: *Mapping ClubEvent to C++*

```

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.

C++ 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

Overview

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 166](#) 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 166: *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 166 on page 363](#) maps to the C++ `SequenceAnyList` class shown in [Example 167](#). 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 167: *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 412](#).

C++ 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 158 on page 343](#)).

```
// 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 168](#) 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 168: 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
    );
```

Example 168:*The `IT_Bus::AnyList` Class*

```

AnyList (
    const size_t min_occurs,
    const size_t max_occurs,
    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&);

```

Example 168:*The `IT_Bus::AnyList` Class*

```
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

Overview

This section describes how to define and use nillable types; that is, XML elements defined with `xsd:nilable="true"`.

In this section

This section contains the following subsections:

Introduction to Nillable Types	page 369
Nillable Atomic Types	page 371
Nillable User-Defined Types	page 375
Nested Atomic Type Nillable Elements	page 378
Nested User-Defined Nillable Elements	page 382
Nillable Elements of an Array	page 387

Introduction to Nillable Types

Overview

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 is a nillable element. If the `nillable` attribute is missing, the default 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 169](#) shows the public member functions of the `IT_Bus::NillableValueBase` class, which provides the C++ API for nillable types.

Example 169: C++ API for Nillable Types

```
// C++
namespace IT_Bus
{
    template <class T>
    class NillableValueBase : public Nillable
    {
    public:
        virtual ~NillableValueBase();
        virtual AnyType& operator=(const AnyType& other);

        virtual Boolean is_nil() const;
        virtual void set_nil();
        ...
        virtual const T&
```

Example 169: C++ API for Nillable Types

```
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

Overview

This subsection describes how to define and use XML schema nillable atomic types. In C++, every atomic type, *AtomicTypeName*, has a nillable counterpart, *AtomicTypeNameNillable*. 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 369](#).

Table of nillable atomic types

[Table 13](#) shows how the XML schema atomic types map to C++ when the `xsd:nillable` flag is set to `true`.

Table 13: *Nillable Atomic Types*

Schema Type	Nillable C++ Type
<code>xsd:anyType</code>	<i>Not supported as nillable</i>
<code>xsd:boolean</code>	<code>IT_Bus::BooleanNillable</code>
<code>xsd:byte</code>	<code>IT_Bus::ByteNillable</code>
<code>xsd:unsignedByte</code>	<code>IT_Bus::UByteNillable</code>
<code>xsd:short</code>	<code>IT_Bus::ShortNillable</code>
<code>xsd:unsignedShort</code>	<code>IT_Bus::UShortNillable</code>
<code>xsd:int</code>	<code>IT_Bus::IntNillable</code>
<code>xsd:unsignedInt</code>	<code>IT_Bus::UIntNillable</code>
<code>xsd:long</code>	<code>IT_Bus::LongNillable</code>
<code>xsd:unsignedLong</code>	<code>IT_Bus::ULongNillable</code>
<code>xsd:float</code>	<code>IT_Bus::FloatNillable</code>
<code>xsd:double</code>	<code>IT_Bus::DoubleNillable</code>
<code>xsd:string</code>	<code>IT_Bus::StringNillable</code>
<code>xsd:QName</code>	<code>IT_Bus::QNameNillable</code>

Table 13: *Nillable Atomic Types*

Schema Type	Nillable C++ Type
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 170 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 170: *WSDL Example Showing Some Nillable Atomic Types*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BaseService"
  targetNamespace="http://soapinterop.org/"
  ...
```

Example 170: 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>
...

```

C++ example

[Example 171](#) shows how to use nillable atomic types, `IT_Bus::StringNillable`, `IT_Bus::ShortNillable`, `IT_Bus::IntNillable`, and `IT_Bus::FloatNillable`, in a simple C++ example.

Example 171: 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 {
    // 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

Overview

This subsection describes how to define and use nillable user-defined types. In C++, every user-defined type, *UserTypeName*, has a nillable counterpart, *UserTypeNameNillable*.

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 369](#).

WSDL example

[Example 172](#) 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 172: 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"/>
        </all>
      </complexType>
      ...
    </schema>
  </types>
  ...
```

C++ mapping

[Example 173](#) 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 173: *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 369](#).

C++ example

The following C++ 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 C++ 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

Overview

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 387](#).

WSDL example

[Example 174](#) defines a sequence complex type, `Nil_SOAPStruct`, which has some nillable elements, `varInt`, `varFloat`, and `varString`.

Example 174: *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>
    <schema targetNamespace="http://soapinterop.org/xsd"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      ...
```

Example 174: WSDL Example of a Sequence Type with Nillable Elements

```

    <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 175 shows how the `Nil_SOAPStruct` sequence complex type is mapped to C++. Note how the accessors for the nillable member elements, `getElementName()`, return a pointer instead of a value; and how the modifiers for the nillable member elements, `setElementName()`, take either a pointer argument or a reference argument. For example, the `getvarInt()` function returns a pointer to an `IT_Bus::Int` rather than an `IT_Bus::Int` value.

Example 175: 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);
  };
}

```

Example 175: C++ Mapping of the *Nil_SOAPStruct* Sequence Type

```

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;
...
};

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;
...
};

```

C++ example

The following C++ 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 C++ 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 `getElementName()` 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

Overview

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 387](#).

WSDL example

[Example 176](#) defines a sequence complex type, `Nil_NestedSOAPStruct`, which includes a nillable element of `SOAPStruct` type, `varSOAP`.

Example 176: *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>
```

Example 176: WSDL Example of a Nilable All Type inside a Sequence Type (Continued)

```

        <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 177](#) 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 177: 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);

```

Example 177: C++ Mapping of the *Nil_NestedSOAPStruct* Type

```

    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, *UserTypeNillablePtr*. The *NillablePtr* template types are similar to the `std::auto_ptr<>` template types from the Standard Template Library—see “Smart Pointers” on page 60.

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 178 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 178: 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();
        ...
    };
}

```


Example 178: *The NillablePtr Template Class (Continued)*

```

        void set(const T* data);

        virtual Boolean is_nil() const;

        virtual const QName& get_type() const;
        ...
    };
    ...
};

```

C++ example

The following C++ 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 C++ 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

Overview

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 179](#) 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 179: 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"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      ...
```

Example 179: WSDL Example of an Array with Nillable Elements

```

        <complexType name="Nil_SOAPArray">
            <sequence>
                <element name="item" nillable="true"
                    type="xsd:short" minOccurs="10"
                    maxOccurs="10"/>
            </sequence>
        </complexType>
        ...
    </schema>
</types>
...

```

C++ mapping

Example 180 shows how the Nil_SOAPArray array complex type is mapped to C++. Note that the array elements are of IT_Bus::ShortNillable type.

Example 180: 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;
    };

```

Example 180: C++ Mapping of the Nil_SOAPArray Array Type (Continued)

```

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;
};

```

C++ 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

Overview

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 25](#) 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>`.

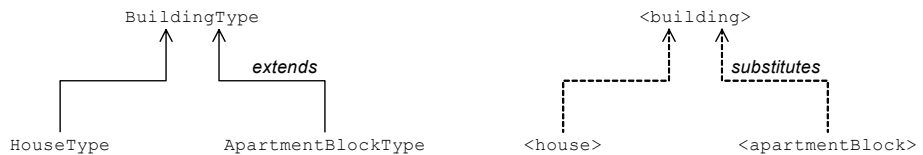


Figure 25: Relationship Between Elements in a Substitution Group

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="xsd:building"/>
    <element name="site" type="xsd:SiteType"/>
  </sequence>
</complexType>
```

Note: Currently, Artix does *not* support substitutable elements in an `<all>` complex type.

XSD example

[Example 181](#) shows the definition of a sequence group, `PropertyType`, that includes a single substitutable element, `xsd:building`.

Example 181: 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">
    <sequence>
      <element name="squareMeters" type="xsd:int"/>
    </sequence>
  </complexType>
```

Example 181: *Sequence Type Containing a Substitutable Element*

```

<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>

```


Example 181: *Sequence Type Containing a Substitutable Element*

```

        <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 182](#) shows how the `PropertyType` sequence group from [Example 181 on page 391](#) maps to C++.

Example 182: *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
        {

```

Example 182: C++ Mapping of PropertyType Sequence Type

```

        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);

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 183](#) redefines `PropertyChoiceType` as a *choice group* that contains a single substitutable element, `xsd1:building`.

Example 183: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>

</schema>
```

The `PropertyChoiceType` choice group defined in the preceding [Example 183](#) maps to the C++ `PropertyChoiceType` class shown in [Example 184](#).

Example 184:C++ Mapping of the PropertyChoiceType Choice Group

```
// C++
namespace COM_IONA_SCHEMAS_REALESTATE
{
  class PropertyChoiceType : public IT_Bus::ChoiceComplexType
  {
```

Example 184: C++ Mapping of the PropertyChoiceType Choice Group

```

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 185](#) contains an unbounded number of `building` elements.

Example 185: 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 350](#).

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 181 on page 391](#) 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

Overview

In addition to the basic array types described in “[Arrays](#)” on page 323, 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.

In this section

This section contains the following subsections:

Introduction to SOAP Arrays	page 401
Multi-Dimensional Arrays	page 405
Sparse Arrays	page 408
Partially Transmitted Arrays	page 411

Introduction to SOAP Arrays

Overview

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](#)
 - [C++ 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 186 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 186: 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>
    </types>
  </definitions>
```

Sample encoding

[Example 187](#) 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 187: 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.

C++ example

[Example 188](#) shows a C++ example of how to allocate and initialize an `ArrayOfSOAPString` instance with four elements.

Example 188: C++ 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 405](#).
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 408](#)). There are no default element values. Uninitialized elements are flagged as empty.

Multi-Dimensional Arrays

Overview

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 189](#) 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 189: Definition of the `Array2OfInt` SOAP Array

```
<definitions ... >
  <types>
    <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 a multi-dimensional SOAP array

[Example 190](#) 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 190: 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.

C++ example of a multi-dimensional SOAP array

[Example 191](#) shows a C++ example of how to allocate and initialize an `Array2OfInt` instance with dimensions, `[2, 3]`.

Example 191: Initializing an `Array2OfInt` SOAP Array

```
// C++
1 const size_t extents2[] = {2, 3};
  Array2OfInt a2_soap(extents2);

  size_t position[2];
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++) {
3       a2_soap[position] = (IT_Bus::Int) (i+1)*(j+1);
    }
  }
```

Example 191: *Initializing an Array2OfInt SOAP Array*

```
}
```

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

Overview

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 192](#) 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 192: *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 193](#) shows an example of how to initialize a sparse array of `Array2OfInt` type.

Example 193: 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 191 on page 406](#)). 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 194 shows an example of how to read a sparse array of `Array2OfInt` type.

Example 194: 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

Overview

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 195](#) shows the encoding of a partially transmitted `ArrayOfSOAPString` instance.

Example 195: *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

Overview

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.

In this section

This section contains the following subsections:

Introduction to IT_Vector	page 413
Summary of IT_Vector Operations	page 416

Introduction to IT_Vector

Overview

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 14](#) are *not* defined in `IT_Vector`.

Table 14: *Member Functions Not Defined in `IT_Vector`*

Function	Type of Operation
<code>at()</code>	Element access (with range check)
<code>clear()</code>	List operation
<code>assign()</code>	Assignment
<code>resize()</code>	Size and capacity
<code>max_size()</code>	

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 196](#) shows how to use these basic vector operations to initialize an integer vector with the first one hundred integer squares.

Example 196: *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 196 on page 414](#) can be written in a more idiomatic style using vector iterators, as shown in [Example 197](#).

Example 197:*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

Overview

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 15](#) lists the member types defined in `IT_Vector<T>`.

Table 15: *Member Types Defined in `IT_Vector<T>`*

Member Type	Description
<code>value_type</code>	Type of element.
<code>size_type</code>	Type of subscripts.
<code>difference_type</code>	Type of difference between iterators.
<code>iterator</code>	Behaves like <code>value_type*</code> .
<code>const_iterator</code>	Behaves like <code>const value_type*</code> .
<code>reverse_iterator</code>	Iterates in reverse, like <code>value_type*</code> .
<code>const_reverse_iterator</code>	Iterates in reverse, like <code>const value_type*</code> .
<code>reference</code>	Behaves like <code>value_type&</code> .
<code>const_reference</code>	Behaves like <code>const value_type&</code> .

Iterators

[Table 16](#) lists the `IT_Vector` member functions returning iterators.

Table 16: *Iterator Member Functions of `IT_Vector<T>`*

Iterator Member Function	Description
<code>begin()</code>	Points to first element.
<code>end()</code>	Points to last element.
<code>rbegin()</code>	Points to first element of reverse sequence.
<code>rend()</code>	Points to last element of reverse sequence.

Element access

[Table 17](#) lists the `IT_Vector` element access operations.

Table 17: *Element Access Operations for `IT_Vector<T>`*

Element Access Operation	Description
<code>[]</code>	Subscripting, unchecked access.
<code>front()</code>	First element.
<code>back()</code>	Last element.

Stack operations

[Table 18](#) lists the `IT_Vector` stack operations.

Table 18: *Stack Operations for `IT_Vector<T>`*

Stack Operation	Description
<code>push_back()</code>	Add to end.
<code>pop_back()</code>	Remove last element.

List operations

[Table 19](#) lists the `IT_Vector` list operations.

Table 19: *List Operations for `IT_Vector<T>`*

List Operations	Description
<code>insert(p, x)</code>	Add <code>x</code> before <code>p</code> .
<code>insert(p, n, x)</code>	Add <code>n</code> copies of <code>x</code> before <code>p</code> .
<code>insert(first, last)</code>	Add elements from <code>[first:last[</code> before <code>p</code> .
<code>erase(p)</code>	Remove element at <code>p</code> .
<code>erase(first, last)</code>	Erase <code>[first:last[</code> .

Other operations

[Table 20](#) lists the other operations supported by `IT_Vector`.

Table 20: *Other Operations for `IT_Vector<T>`*

Operation	Description
<code>size()</code>	Number of elements.
<code>empty()</code>	Is the container empty?
<code>capacity()</code>	Space allocated.
<code>reserve()</code>	Reserve space for future expansion.
<code>swap()</code>	Swap all the elements between two vectors.
<code>==</code>	Test vectors for equality (member-wise).

Unsupported XML Schema Constructs in Artix

Overview

The following XML schema constructs are currently not supported in Artix:

- Built-in types:
 - ◆ `xs:duration`
 - ◆ `xs:NOTATION`
 - ◆ `xs:IDREF`
 - ◆ `xs:IDREFS`
 - ◆ `xs:ENTITY`
 - ◆ `xs:ENTITIES`
- `xs:redefine`
- `xs:notation`
- `xs:simpleType`
 - ◆ All facets except for enumeration.
 - ◆ final attribute.
- `xs:complexType`
 - ◆ mixed, final, block, and abstract attributes.
 - ◆ simpleContent/restriction.
 - ◆ complexContent/restriction.
- `xs:element`
 - ◆ final, block, fixed, default and abstract attributes.
- `xs:attribute`
 - ◆ global attributes.
 - ◆ ref attribute.
 - ◆ form attribute.
- `xs:group`
 - ◆ minOccurs, maxOccurs on local groups.
 - ◆ all inside a group.
- `xs:anyAttribute`
- `xs:anySimpleType`

- `xs:unique`
- `xs:key`
- `xs:keyref`
- `xs:selector`
- `xs:field`
- `id` attribute on schema constructs, wherever it is applicable.

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).

In this chapter

This chapter discusses the following topics:

Introduction to IDL Mapping	page 422
IDL Basic Type Mapping	page 424
IDL Complex Type Mapping	page 426
IDL Module and Interface Mapping	page 435

Introduction to IDL Mapping

Overview

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
```

For a detailed discussion of these command-line utilities, see the *Artix User's Guide*.

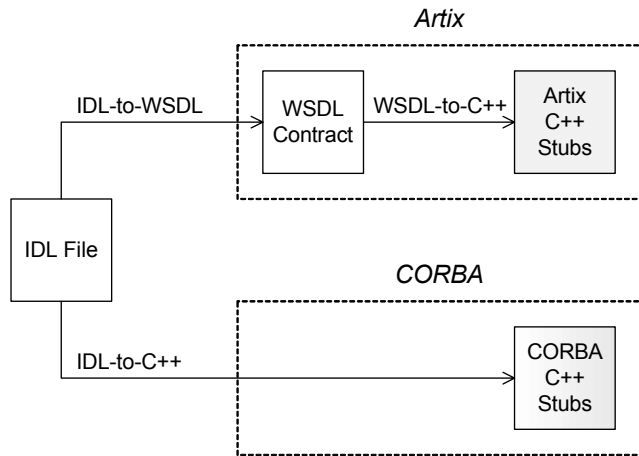
Alternative C++ mappings

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 an IONA-proprietary mapping.
- CORBA IDL-to-C++ mapping—as specified in the [OMG C++ Language Mapping document](http://www.omg.org) (<http://www.omg.org>). This mapping is used, for example, by the IONA's Orbix.

These alternative approaches are illustrated in [Figure 26](#).

Figure 26: *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 also 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

Overview

Table 21 shows how IDL basic types are mapped to WSDL and then to C++.

Table 21: 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
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	<i>Not supported</i>	<i>Not supported</i>
octet	xsd:unsignedByte	IT_Bus::UByte
fixed	xsd:decimal	IT_Bus::Decimal
Object	references:Reference	IT_Bus::Reference

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

Overview

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 275](#).

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 `getUnionMember()` and `setUnionMember()` 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 292](#).

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 `getStructMember()` and `setStructMember()` 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 289](#).

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="xsdl: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 323](#) and [“IT_Vector Template Class” on page 412](#).

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 323](#) and [“IT_Vector Template Class” on page 412](#).

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="xsdl:SampleTypes.GenericExc"/>
...
<message name="_exception.SampleTypes.GenericExc">
  <part name="exception"
    element="xsdl: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 47](#).

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

Overview

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 Stub and Skeleton Code” on page 2](#)). 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 `IT_Bus::Reference` 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(
    IT_Bus::Reference & 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 `IT_Bus::Reference` object into the `FooClient` constructor.

See [“Artix References” on page 103](#) for more details.

Operation mapping

[Example 198](#) 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 198: 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 198 on page 437](#), map to C++ as shown in [Example 199](#),

Example 199: 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 200](#) 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 200: Example IDL Attributes

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

The attributes from the preceding IDL, [Example 200 on page 438](#), map to C++ as shown in [Example 201](#),

Example 201: *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.

In this chapter

This chapter discusses the following topics:

Introduction to Reflection	page 442
The IT_Bus::Var Template Type	page 445
Reflection API	page 449
Reflection Example	page 476

Introduction to Reflection

Overview

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 450](#) 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 202: 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
    };
};
```

Example 202: *Definition of the `IT_Bus::AnyType::Kind` Enumeration*

```

        SIMPLE,           // simpleType restriction
        SEQUENCE,
        ALL,
        CHOICE,
        SIMPLE_CONTENT,
        ELEMENT_LIST,
        SOAP_ENC_ARRAY,
        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 476](#)) or to convert an Artix data type into another data format.

The IT_Bus::Var Template Type

Overview

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.

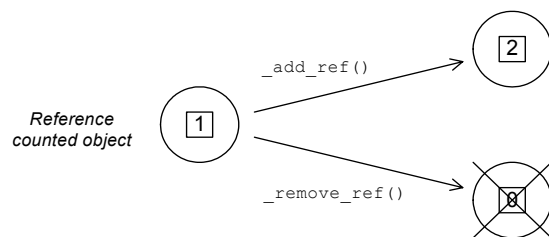
Reference counted objects

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 27 illustrates how the reference count is affected by the `_add_ref()` and `_remove_ref()` functions.

Figure 27: Reference Counted Object



Var template class

Table 22 shows the basic operations supported by the `IT_Bus::Var<T>` template class.

Table 22: *Basic `IT_Bus::Var<T>` Operations*

Operation	Description
=	The assignment operator distinguishes between the following kinds of assignment: <ul style="list-style-type: none"> • 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, `f_v`.

```
// C++
#include <it_bus/var.h>
...
{
    Foo* plain_p = new Foo();           // Initially, ref count = 1

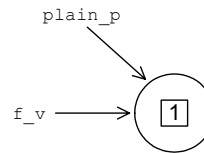
    // Assign the plain pointer, plain_p, to the Var, f_v
    IT_Bus::Var<Foo> f_v = plain_p; // Ref count = 1

    // 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 28 shows the state of the variables in the preceding example just after the assignment to the Var, `f_v`.

Figure 28: 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

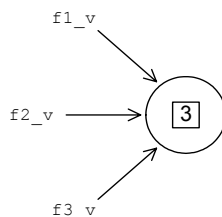
    IT_Bus::Var<Foo> f2_v = f1_v;      // Ref count = 2
    IT_Bus::Var<Foo> f3_v = f1_v;      // Ref count = 3

    // 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 29 shows the state of the variables in the preceding example just after the assignment to the Var, `f3_v`.

Figure 29: 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 476.

Reflection API

Overview

This section briefly describes the Artix reflection API. The header files for the classes described in this section are located in

ArtixInstallDir/artix/Version/include/it_bus/reflect.

In this section

This section contains the following subsections:

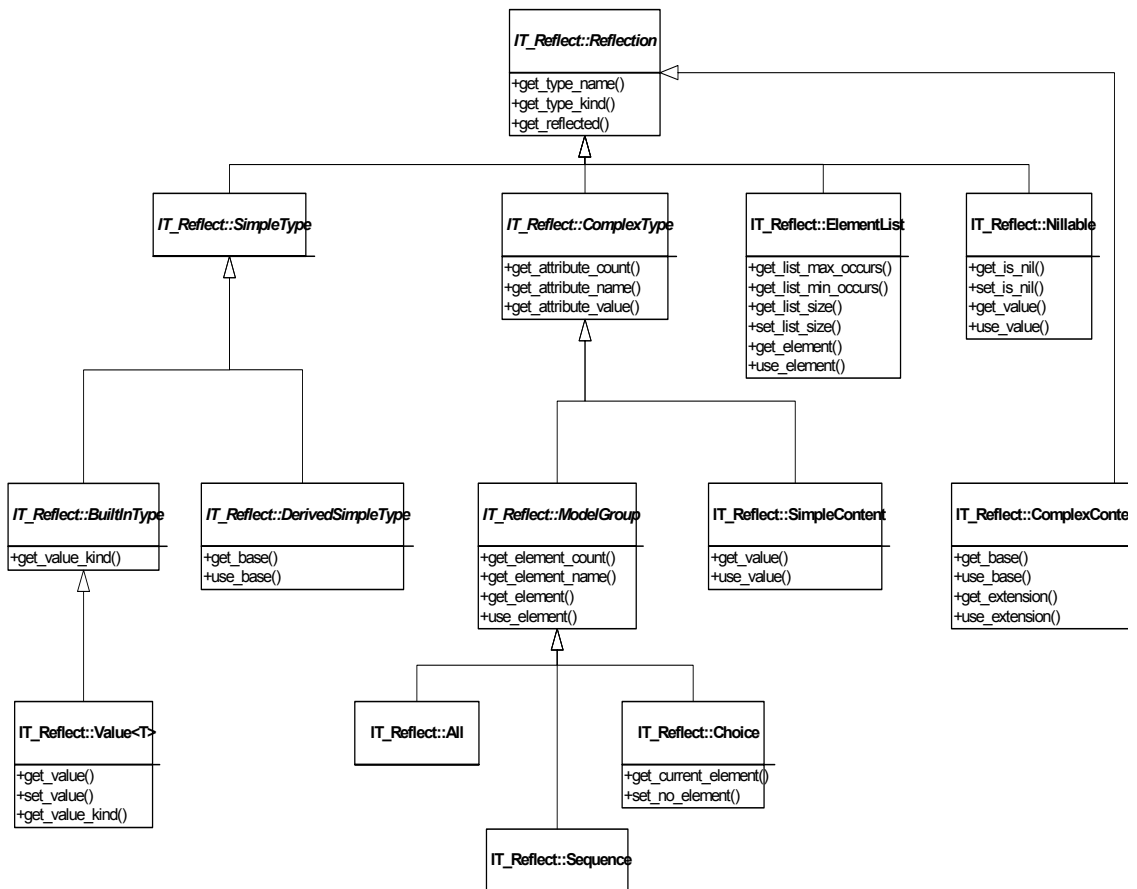
Overview of the Reflection API	page 450
IT_Reflect::Value<T>	page 452
IT_Reflect::All	page 456
IT_Reflect::Sequence	page 459
IT_Reflect::Choice	page 462
IT_Reflect::SimpleContent	page 465
IT_Reflect::ComplexContent	page 467
IT_Reflect::ElementList	page 470
IT_Reflect::SimpleTypeList	page 472
IT_Reflect::Nillable	page 473

Overview of the Reflection API

Overview

Artix provides a collection of reflection classes to parse the contents of XML schema data objects. [Figure 30](#) gives an overview of the inheritance hierarchy for this C++ reflection API.

Figure 30: Reflection API Inheritance Hierarchy



Base classes

The following classes in [Figure 30 on page 450](#) are used as base classes:

<code>IT_Reflect::Reflection</code>	Base class for all reflection classes.
<code>IT_Reflect::SimpleType</code>	Base class for all built-in and restricted simple types.
<code>IT_Reflect::BuiltInType</code>	Base class for all built-in types.
<code>IT_Reflect::ComplexType</code>	Base class for all complex types (types with attributes) except <code>ComplexContent</code> .
<code>IT_Reflect::ModelGroup</code>	Base class for <code>xsd:all</code> , <code>xsd:sequence</code> and <code>xsd:choice</code> types.

Leaf classes

The following classes in [Figure 30 on page 450](#) are the leaf classes for the reflection API:

<code>IT_Reflect::Value<T></code>	Template class for built-in types.
<code>IT_Reflect::DerivedSimpleType</code>	Reflection class for restricted simple types.
<code>IT_Reflect::All</code>	Reflection class for the <code>xsd:all</code> type.
<code>IT_Reflect::Sequence</code>	Reflection class for the <code>xsd:sequence</code> type.
<code>IT_Reflect::Choice</code>	Reflection class for the <code>xsd:choice</code> type.
<code>IT_Reflect::SimpleContent</code>	Reflection class for <code>xsd:simpleContent</code> types.
<code>IT_Reflect::ComplexContent</code>	Reflection class for <code>xsd:complexContent</code> types.
<code>IT_Reflect::ElementList</code>	Reflection class representing an element declared with non-default <code>minOccurs</code> or non-default <code>maxOccurs</code> properties.
<code>IT_Reflect::Nillable</code>	Reflection class representing an element declared with <code>nillable="true"</code> .

IT_Reflect::Value<T>

Overview

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 203](#) shows an example of schema element defined to be of simple type, `xsd:string`.

Example 203: 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 204](#) 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 204: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);

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 2 on page 247](#).

Other built-in types

For the list of supported non-atomic types, see [Table 23](#).

Table 23: *Non-Atomic Built-In Types Supported by Reflection*

Value Kind	Schema Type	C++ Type
ANYURI	<code>xsd:anyURI</code>	<code>IT_Bus::AnyURI</code>
ANY	<code>xsd:any</code>	<code>IT_Bus::Any</code>
ANY_LIST	<code>xsd:any</code> (<i>multiply occurring</i>)	<code>IT_Bus::AnyList</code>
ANY HOLDER	<code>xsd:anyType</code>	<code>IT_Bus::AnyHolder</code>
REFERENCE	<code>references:Reference</code>	<code>IT_Bus::Reference</code>

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

Overview

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 205](#) shows a sample schema for an `xsd:all` type.

Example 205: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 206](#) shows the `IT_Reflect::All` member functions, which enable you to access and modify the contents and attributes of an `xsd:all` type.

Example 206: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();
```


Example 206:*IT_Reflect::All Member Functions*

```

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));

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(());

```

Example 206:*IT_Reflect::All Member Functions*

```
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

Overview

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 207](#) shows a sample schema for an `xsd:sequence` type.

Example 207: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 208 shows the `IT_Reflect::Sequence` member functions, which enable you to access and modify the contents and attributes of an `xsd:sequence` type.

Example 208: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
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*
```

Example 208:*IT_Reflect::Sequence Member Functions*

```

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

Overview

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 209](#) shows a sample schema for an `xsd:choice` type.

Example 209: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 210](#) shows the `IT_Reflect::Choice` member functions, which enable you to access and modify the contents and attributes of an `xsd:choice` type.

Example 210: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
```

Example 210:*IT_Reflect::Choice Member Functions*

```

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));

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

```

Example 210:*IT_Reflect::Choice Member Functions*

```
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

Overview

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 310](#) for more details).

This subsection discusses the following topics:

- [Sample schema](#)
- [IT_Reflect::SimpleContent member functions](#)

Sample schema

[Example 211](#) shows a sample schema for an `xsd:simpleContent` type.

Example 211: 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 212](#) shows the `IT_Reflect::SimpleContent` member functions, which enable you to access and modify the contents and attributes of an `xsd:simpleContent` type.

Example 212:*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&
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

Overview

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 313](#) for more details).

This subsection discusses the following topics:

- [Sample schema](#)
- [IT_Reflect::ComplexContent member functions](#)

Sample schema

[Example 213](#) shows a sample schema for an `xsd:complexContent` type.

Example 213: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 214 shows the `IT_Reflect::SimpleContent` member functions, which enable you to access and modify the contents and attributes of an `xsd:complexContent` type.

Example 214:*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(());
```

Example 214:*IT_Reflect::ComplexContent Member Functions*

```
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

Overview

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`:

<code>minOccurs="1" maxOccurs="1"</code>	Returns the element directly.
<i>All other values</i>	Returns <code>IT_Reflect::ElementList</code> .

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 215](#) shows a sample schema for an Artix array, which is represented as an element list.

Example 215: 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 216](#) shows the `IT_Reflect::ElementList` member functions, which enable you to access and modify the contents of an Artix array type.

Example 216:*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

Overview

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 217: *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

Overview

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 24: *Effect of nillable, minOccurs and maxOccurs Settings*

nillable	minOccurs/maxOccurs	Return Value
nillable="false"	minOccurs="1" maxOccurs="1"	Returns the element directly.
nillable="false"	<i>All other values</i>	Returns <code>IT_Reflect::ElementList</code> .
nillable="true"	minOccurs="1" maxOccurs="1"	Returns <code>IT_Reflect::Nillable</code> containing an element directly.
nillable="true"	<i>All other values</i>	Returns an <code>IT_Reflect::ElementList</code> containing a list of <code>IT_Reflect::Nillables</code> .

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 218](#) shows a sample schema for a sequence type with nillable elements.

Example 218:*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 219](#) shows the `IT_Reflect::Nillable` member functions, which enable you to access and modify the contents of a nillable type.

Example 219:*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();
```

Example 219:*IT_Reflect::Nilable Member Functions*

```
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));
```

Reflection Example

Overview

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.

In this section

This section contains the following subsections:

Print an <code>IT_Bus::AnyType</code>	page 477
Print Atomic and Simple Types	page 482
Print Sequence, Choice and All Types	page 488
Print SimpleContent Types	page 491
Print ComplexContent Types	page 493
Print Multiple Occurrences	page 496
Print Nillables	page 498

Print an `IT_Bus::AnyType`

Overview

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 220](#) 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 220: *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>
#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;
```

Example 220: Code Example for Printing an `IT_Bus::AnyType` Instance

```

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());
}

```

Example 220: Code Example for Printing an `IT_Bus::AnyType` Instance

```

Printer::Printer()
:
  m_indent(0),
  m_in_list(IT_FALSE)
{
}

Printer::~Printer()
{
}

void
5 Printer::print(
  const IT_Reflect::Reflection* reflection
)
{
6   assert(reflection != 0);
  switch (reflection->get_type_kind())
  {
    case AnyType::BUILT_IN:
      print(IT_DYNAMIC_CAST(const IT_Reflect::BuiltInType*,
7 reflection));
      break;
    case AnyType::SIMPLE:
      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:

```

Example 220: Code Example for Printing an `IT_Bus::AnyType` Instance

```

        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 445](#) 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

Overview

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 221](#) 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 221: *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
)
{
```

Example 221: Code Example for Printing Atomic and Simple Types

```

    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;
    }

void
7   Printer::print(
        const IT_Reflect::BuiltInType* data
    )
    {
8   assert(data != 0);
        switch (data->get_value_kind())
        {

```

Example 221: Code Example for Printing Atomic and Simple Types

```

    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;
    }

void
7   Printer::print(
        const IT_Reflect::BuiltInType* data
    )
    {
8   assert(data != 0);
        switch (data->get_value_kind())
        {

```

Example 221: Code Example for Printing Atomic and Simple Types

9

```

case IT_Reflect::BuiltInType::BOOLEAN:
    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:

```

Example 221: Code Example for Printing Atomic and Simple Types

```

    case IT_Reflect::BuiltInType::BASE64BINARY:
    case IT_Reflect::BuiltInType::DATE:
    case IT_Reflect::BuiltInType::TIME:
    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 275](#) 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

Overview

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 222](#) 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 222: 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 indenter(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() <<
            ": ";
```


Example 222: Code Example for Printing Sequence and All Types

```

8      Indenter indent(this);
        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 493](#) 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 445](#) 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 223](#) shows a code extract from the `Printer` class, which shows the function for printing XML choice types using the Artix Reflection API.

Example 223: *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 445. In this case, the returned pointer is of `IT_Reflect::Reflection*` type.

Print SimpleContent Types

Overview

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 224](#) 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 224: 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 493](#) 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

Overview

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 313](#)).

Code extract

[Example 225](#) 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 225: 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());
    }
}
```

Example 225: Code Example for Printing ComplexContent Types

```

}

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)
        {
            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 313](#) 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

Overview

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 `minOccurs=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 226](#) 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 226: 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

Overview

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 227](#) shows a code extract from the `Printer` class, which shows the function for printing nillables using the Artix reflection API.

Example 227: 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).

In this chapter

This chapter discusses the following topics:

Introduction to Persistent Maps	page 502
Creating a Persistent Map	page 505
Inserting, Extracting and Removing Data	page 508
Handling Exceptions	page 512
Supporting High Availability	page 515
Configuration Example	page 518

Introduction to Persistent Maps

Overview

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 518](#). 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

Overview

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 persistent 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 228](#) shows you how to create a string persistent map, `f_map`, that uses `float` as its data type.

Example 228: 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 229](#) shows you how to create a persistent map, `i_map`, that uses `char` as its key type and `int` as its data type.

Example 229: 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 an 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 503](#) for the complete list of persistence handler types.

[Example 230](#) shows you how to create two persistent maps using the `PersistentMapBase` template: the `QtOMap` type maps `QNames` to `IT_Bus::Reference` instances and the `ChartoWSDLMap` type maps chars to instances of a user complex type, `MyWSDLType`.

Example 230: Creating a Persistent Map for Complex Types

```
// C++
using namespace IT_Bus;

typedef IT_Bus::PersistentMapBase<
    IT_Bus::QName,
    IT_Bus::Reference,
    IT_Bus::QNamePersistenceHandler,
    IT_Bus::AnyTypePersistenceHandler<IT_Bus::Reference>
> QtOMap;

typedef IT_Bus::PersistentMapBase<
    char,
    MyWSDLType,
    IT_Bus::PODPersistenceHandler,
    IT_Bus::AnyTypePersistenceHandler<MyWSDLType>
> ChartoWSDLMap;

DBConfig cfg(bus);
QtOMap map("myRefMap", &cfg);
ChartoWSDLMap myMap("myDataMap", &cfg);
```

Inserting, Extracting and Removing Data

Overview

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 229 on page 506](#)), 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

Overview

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 231](#) shows how Artix database exceptions should be handled on the server side for applications that use the persistent map API.

Example 231: 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)
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 232](#) shows the signatures of the member functions from the `IT_Bus::DBException` class.

Example 232:*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 514](#)). 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 233: 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

Overview

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 (as of Artix 3.0.2) 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 *Deploying and Managing 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 234](#) 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 234: 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,

```

Example 234: *Write-Request Forwarding Example*

```
        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 412](#)).
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/artix/Version/demos/advanced/high_availability_persistent_servers
```

Configuration Example

Overview

[Example 235](#) shows the minimal configuration that is required to configure persistence based on the Berkeley DB.

Example 235: 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:

<code>plugins:artix:db:env_name</code>	Specifies the filename for the Berkeley DB environment file. It can be any string and can have any file extension (for example, <code>myDB.env</code>).
<code>plugins:artix:db:home</code>	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 *Deploying and Managing Artix Solutions*.

Transactions in Artix

This chapter discusses the Artix support for distributed transaction processing.

In this chapter

This chapter discusses the following topics:

Introduction to Transactions	page 520
Selecting the Transaction System	page 525
Transaction API	page 536
Transaction Demarcation	page 539
Participants and Resources	page 542
Threading	page 552
Transaction Propagation	page 556
Notification Handlers	page 560
Reliable Messaging with MQ Transactions	page 562
Client Example	page 571

Introduction to Transactions

Overview

This section gives a short overview of the main features supported by Artix transactions. The Artix transaction API is designed to be compatible with a variety of different underlying transaction systems. Generally, you can access the transaction system using a technology-neutral API, but the technology-specific APIs are also available, in case you need to access more advanced functionality.

The main features of Artix transactions are as follows:

- [Supported protocols](#)
- [Client-side transaction support](#)
- [Server-side transaction support](#)
- [Compatibility with Orbix](#)
- [Pluggable transaction system](#)
- [One-phase commit](#)
- [Two-phase commit](#)
- [Transaction propagation](#)

Supported protocols

Artix supports distributed transactions using the following protocols:

- CORBA binding over IIOP
- SOAP binding over any compatible transport

Client-side transaction support

Transaction demarcation functions (`begin_transaction()`, `commit_transaction()` and `rollback_transaction()`) can be used on the client side to initiate and terminate a transaction. While the transaction is active, all of the operations called from the current thread are included in the transaction (that is, the operations' request headers include a transaction context).

Server-side transaction support

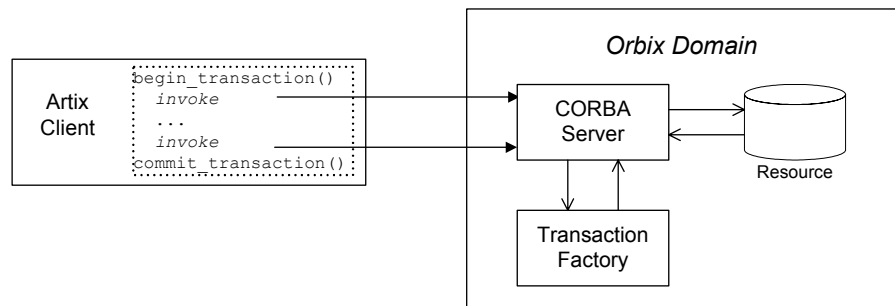
On the server side, an API is provided that enables you to implement *transaction participants* (sometimes referred to as transactional resources). Using transaction participants, you can implement servers that participate in a distributed transaction with the ACID transaction properties (*Atomicity, Consistency, Integrity, and Durability*).

Artix supports several different approaches to implementing a transaction participant, depending on what kind of transaction system is loaded into your application. For example, you might take a technology-neutral approach by implementing the `IT_Bus::TransactionParticipant` class, or you might decide to exploit the special features of a particular transaction system instead.

Compatibility with Orbix

The Artix transaction facility is fully compatible with CORBA OTS in Orbix. Hence, if you already have a transactional server implemented with Orbix ASP, you can easily integrate this with an Artix client, as shown in [Figure 31](#).

Figure 31: *Artix Client Invokes a Transactional Operation on a CORBA OTS Server*



Pluggable transaction system

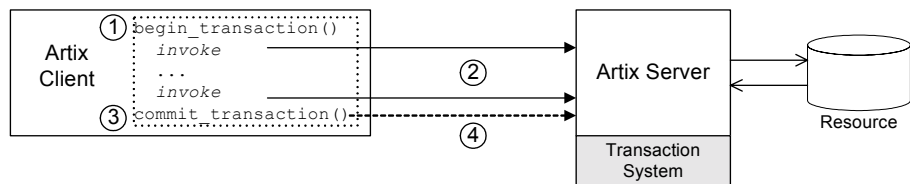
The underlying transaction system used by Artix can be replaced within a pluggable framework. Currently, the following transaction systems are supported by Artix:

- OTS Lite
- OTS Encina
- WS-AtomicTransactions

One-phase commit

Artix supports the one-phase commit (1PC) protocol for transactions. This protocol can be used if there is only one resource participating in the transaction. The 1PC protocol essentially delegates the transaction completion to the single resource manager. [Figure 32](#) shows a schematic overview of the 1PC protocol for a simple client-server system.

Figure 32: *One-Phase Commit Protocol*



The 1PC protocol progresses through the following stages:

1. The client calls `begin_transaction()` to initiate the transaction.
2. Within the transaction, the client calls one or more WSDL operations on the remote server. The WSDL operations are transactional, requiring updates to a persistent resource.
3. The client calls `commit_transaction()` to make permanent any changes caused during the transaction (alternatively, the client could call `rollback_transaction()` to abort the transaction).
4. The transaction system performs the commit phase by sending a notification to the server that it should perform a 1PC commit.

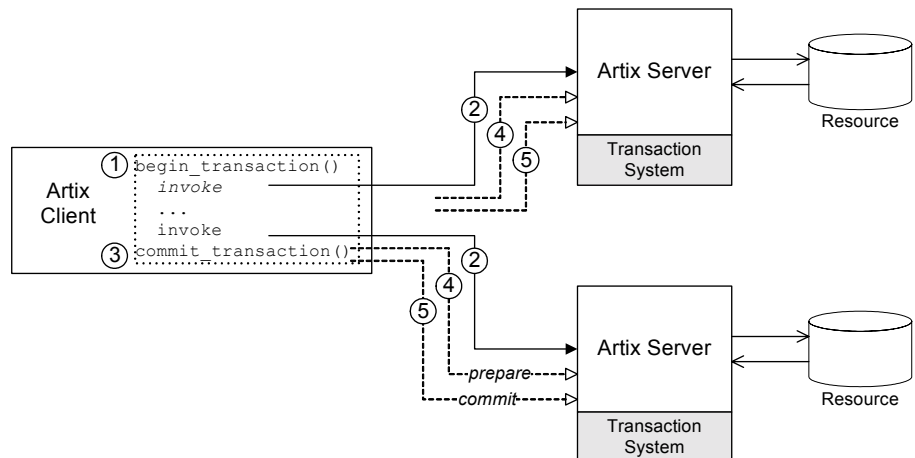
Two-phase commit

The two-phase commit (2PC) protocol enables multiple resources to participate in a transaction. In order to preserve the essential properties of a transaction involving multiple distributed resources, it is necessary to use a more elaborate algorithm. The 2PC algorithm consists of the following two phases:

- *Prepare phase*—the transaction system notifies all of the participants to prepare the transaction. The participants prepare the transaction by saving the information that would be required to redo or undo the changes made during the transaction. At the end of this phase, the participants vote whether to commit or roll back the transaction.
- *Commit (or rollback) phase*—if all of the participants vote to commit the transaction, the transaction system notifies the participants to commit the changes. On the other hand, if one or more participants vote to roll back the transaction, the transaction system notifies the participants to roll back the changes.

Figure 33 shows a schematic overview of the 2PC protocol for a client and two remote servers.

Figure 33: *Two-Phase Commit Protocol*



The 2PC protocol progresses through the following stages:

1. The client calls `begin_transaction()` to initiate the transaction.
2. Within the transaction, the client calls one or more WSDL operations on both of the remote servers.
3. The client calls `commit_transaction()` to make permanent any changes caused during the transaction (alternatively, the client could call `rollback_transaction()` to abort the transaction).
4. The transaction system performs the prepare phase by polling all of the remote transaction participants (the first phase of a two-phase commit).
5. The transaction system performs the commit or rollback phase by sending a notification to all of the remote transaction participants (the second phase of a two-phase commit).

Transaction propagation

If you have a section of code executing within a transaction context, Artix automatically propagates a transaction context with the request message, whenever a remote operation is called.

For example, consider a three-tier system, where a client initiates a transaction, invokes an operation on server 1, and then server 1 makes a further call on server 2. In this scenario, Artix automatically propagates the transaction to server 2. The transaction is propagated, even if the protocol between the client and server 1 differs from the protocol used between server 1 and server 2.

Selecting the Transaction System

Overview

Using the Artix plug-in architecture, you can choose between a number of different transaction system implementations. Because the Artix transaction API is designed to be independent of the underlying transaction system, it is possible to select a particular transaction system at runtime (by specifying the appropriate configuration). Typically, you would choose the transaction system that provides the best match for your services. For example, if the majority of your services are SOAP-based, you would probably select the WS-AT transaction system.

This section describes how to configure an application to use each of the transaction systems supported by Artix.

In this section

This section contains the following subsections:

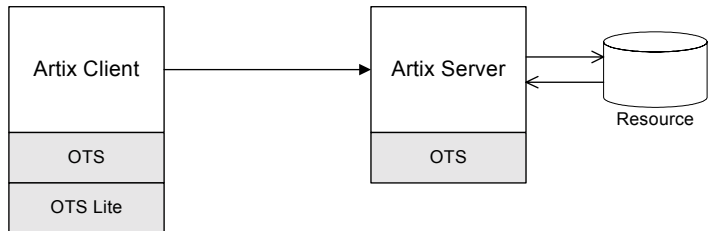
Configuring OTS Lite	page 526
Configuring OTS Encina	page 529
Configuring WS-AT	page 533

Configuring OTS Lite

Overview

The *OTS Lite plug-in* is a lightweight transaction manager, which is subject to the following restrictions: it supports the 1PC protocol only and it lets you register only one resource. This plug-in allows applications that only access a single transactional resource to use the OTS APIs without incurring a large overhead, but allows them to migrate easily to the more powerful 2PC protocol by switching to a different transaction manager. [Figure 34](#) shows a client-server deployment that uses the OTS Lite plug-in.

Figure 34: Overview of a Client-Server System that Uses OTS Lite



Default transaction provider

The following variable specifies the default transaction system used by an Artix client or server:

```
plugins:bus:default_tx_provider:plugin
```

To select the CORBA OTS transaction system, you must initialize this configuration variable with the value, `ots_tx_provider`.

Loading the OTS plug-in

In order to use the CORBA OTS transaction system, the OTS plug-in must be loaded both by the client and by the server. To load the OTS plug-in, include the `ots` plug-in name in the `orb_plugins` list. For example:

```
# Artix Configuration File
ots_lite_client_or_server {
  plugins:bus:default_tx_provider:plugin = "ots_tx_provider";
  orb_plugins = [ ..., "ots"];
};
```

Loading the OTS Lite plug-in

The OTS Lite plug-in, which is capable of managing 1PC transactions, can be loaded on the client side, but it is not usually needed on the server side. You can load the OTS Lite plug-in in one of the following ways:

- *Dynamic loading*—configure Artix to load the `ots_lite` plug-in dynamically, if it is required. For this approach, you need to configure the `initial_references:TransactionFactory:plugin` variable as follows:

```
# Artix Configuration File
ots_lite_client_or_server {
  plugins:bus:default_tx_provider:plugin= "ots_tx_provider";
  orb_plugins = [ ..., "ots"];
  initial_references:TransactionFactory:plugin = "ots_lite";
  ...
};
```

This style of configuration has the advantage that the OTS Lite plug-in is loaded only if it is actually needed.

- *Explicit loading*—load the `ots_lite` plug-in by adding it to the list of `orb_plugins`, as follows:

```
# Artix Configuration File
ots_lite_client {
  plugins:bus:default_tx_provider:plugin= "ots_tx_provider";
  orb_plugins = [ ..., "ots", "ots_lite"];
  ...
};
```

Sample configuration

The following example shows a sample configuration for using the OTS Lite transaction manager:

```
# Artix Configuration File

# Basic configuration for transaction plug-ins (shared library
# names and so on) included in the global configuration scope.
# ... (not shown)

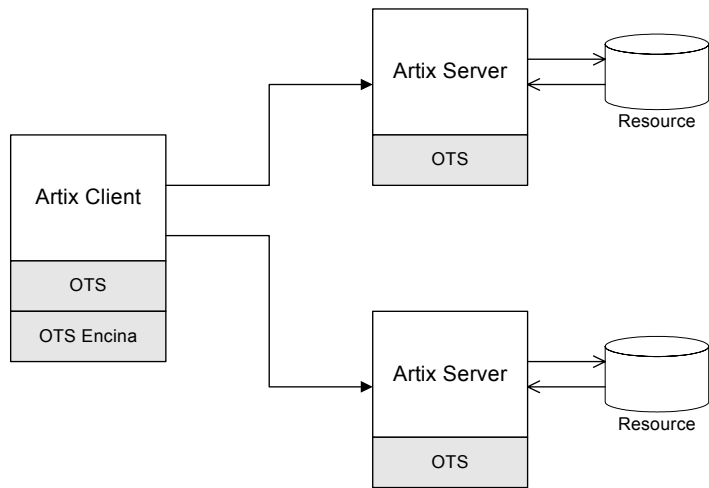
ots_lite_client_or_server {
    plugins:bus:default_tx_provider:plugin= "ots_tx_provider";
    orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop",
"iiop", "ots"];
    initial_references:TransactionFactory:plugin = "ots_lite";
};
```


Configuring OTS Encina

Overview

The Encina OTS Transaction Manager provides full recoverable 2PC transaction coordination implemented on top of the industry proven Encina Toolkit from IBM/Transarc. Encina supports both 1PC and 2PC protocols and allows you to register multiple resources. [Figure 35](#) shows a client/server deployment that uses the OTS Encina plug-in.

Figure 35: Overview of a Client-Server System that Uses OTS Encina



Default transaction provider

The following variable specifies the default transaction system used by an Artix client or server:

```
plugins:bus:default_tx_provider:plugin
```

To select the CORBA OTS transaction system, you must initialize this configuration variable with the value, `ots_tx_provider`.

Loading the OTS plug-in

For applications that use the CORBA OTS transaction system, the OTS plug-in must be loaded both by the client and by the server. To load the OTS plug-in, include the `ots` plug-in name in the `orb_plugins` list. For example:

```
# Artix Configuration File
ots_encina_client_or_server {
  plugins:bus:default_tx_provider:plugin = "ots_tx_provider";
  orb_plugins = [ ..., "ots"];
};
```

Loading the OTS Encina plug-in

The OTS Encina plug-in, which is capable of managing 1PC and 2PC transactions, can be loaded on the client side, but it is not usually needed on the server side. You can load the OTS Encina plug-in in one of the following ways:

- *Dynamic loading*—configure Artix to load the `ots_encina` plug-in dynamically, if it is required. For this approach, you need to configure the `initial_references:TransactionFactory:plugin` variable as follows:

```
# Artix Configuration File
ots_encina_client_or_server {
  plugins:bus:default_tx_provider:plugin="ots_tx_provider";
  orb_plugins = [ ..., "ots"];
  initial_references:TransactionFactory:plugin="ots_encina";
  ...
};
```

This style of configuration has the advantage that the OTS Encina plug-in is loaded only if it is actually needed.

- *Explicit loading*—load the `ots_encina` plug-in by adding it to the list of `orb_plugins`, as follows:

```
# Artix Configuration File
ots_lite_client {
  plugins:bus:default_tx_provider:plugin= "ots_tx_provider";
  orb_plugins = [ ..., "ots", "ots_encina"];
  ...
};
```

Sample configuration

Example 236 shows a complete configuration for using the OTS Encina transaction manager:

Example 236: *Sample Configuration for OTS Encina Plug-In*

```

# Artix Configuration File
ots_lite_client_or_server {
1   plugins:bus:default_tx_provider:plugin= "ots_tx_provider";
   orb_plugins = [ ..., "ots"];

2   initial_references:TransactionFactory:plugin = "ots_encina";

3   plugins:ots_encina:direct_persistence = "true";

4   plugins:ots_encina:initial_disk = "../log/encina.log";
5   plugins:ots_encina:initial_disk_size = "1";
6   plugins:ots_encina:restart_file =
   "../log/encina_restart";
7   plugins:ots_encina:backup_restart_file =
   "../log/encina_restart.bak";

   # Boilerplate configuration settings for OTS Encina:
   # (you should never need to change these)
8   plugins:ots_encina:shlib_name = "it_ots_encina";
   plugins:ots_encina_adm:shlib_name = "it_ots_encina_adm";
   plugins:ots_encina_adm:grammar_db =
   "ots_encina_adm_grammar.txt";
   plugins:ots_encina_adm:help_db = "ots_encina_adm_help.txt";
};

```

The preceding configuration can be described as follows:

1. These two lines configure Artix to use the CORBA OTS transaction system and load the OTS plug-in.
2. This line configures Artix to load the `ots_encina` plug-in dynamically, if it is needed by the application (typically needed on the client side).
3. Configuring Encina to use direct persistence means that the Encina transaction manager service listens on a fixed IP port.
4. The `plugins:ots_encina:initial_disk` variable specifies the path for the initial file used by the Encina OTS for its transaction logs. If this file does not exist when you start the client, Encina OTS automatically creates it (cold start).

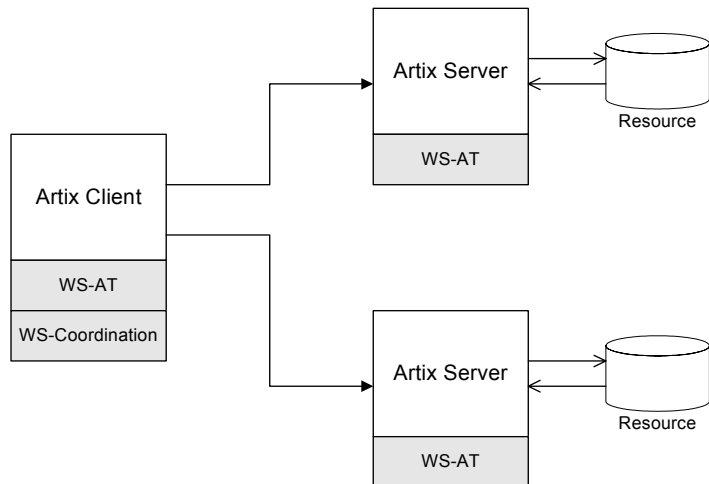
5. The `plugins:ots_encina:initial_disk_size` variable specifies the size of the initial file used by the Encina OTS for its transaction logs. Defaults to 2.
6. The `plugins:ots_encina:restart_file` variable specifies the path for the restart file, which Encina OTS uses to locate its transaction logs. If this file does not exist when you start the client, Encina OTS automatically creates it (cold start).
7. The `plugins:ots_encina:backup_restart_file` variable specifies the path for the backup restart file, which Encina OTS uses to locate its transaction logs. If this file does not exist when you start the client, Encina OTS automatically creates it (cold start).
8. The settings in the next few lines specify the basic configuration of the OTS Encina plug-in. It should not be necessary ever to change the values of these configuration settings.

Configuring WS-AT

Overview

The WS-AtomicTransactions (WS-AT) transaction system uses SOAP headers to transmit transaction contexts between the participants in a transaction. The WS-AT transaction system supports the 2PC protocol and allows you to register multiple resources; unlike OTS Encina, however, it does not support recovery. [Figure 36](#) shows a client/server deployment that uses the WS-AT transaction system.

Figure 36: Overview of a Client-Server System that Uses WS-AT



Default transaction provider

The following variable specifies the default transaction system used by an Artix client or server:

```
plugins:bus:default_tx_provider:plugin
```

To select the WS-AT transaction system, you must initialize this configuration variable with the value, `wsat_tx_provider`.

Plug-ins for WS-AT

The division of the WS-AT transaction system into separate plug-ins reflects the fact that the WS-AT specification has two distinct parts: WS-AtomicTransactions and WS-Coordination.

The following plug-ins are required to support the WS-AT transaction system:

- `wsat_protocol` plug-in—implements WS-AtomicTransactions. It is required by all services and clients that use WS-AT transactions. This plug-in enables an Artix executable to receive and transmit WS-AT transaction contexts.
- `ws_coordination_service` plug-in—implements WS-Coordination. Only one instance of this plug-in is required (typically, loaded into a client). This plug-in coordinates the two-phase commit protocol.

Sample configuration

[Example 237](#) shows a complete configuration for using the WS-AT transaction manager:

Example 237: Sample Configuration for WS-AtomicTransactions

```
# Artix Configuration File
ws_atomic_transactions {
  client
  {
1     orb_plugins = ["local_log_stream",
2     "ws_coordination_service"];
    plugins:bus:default_tx_provider:plugin = "wsat_tx_provider";
  };

  server
  {
3     orb_plugins = ["local_log_stream", "wsat_protocol",
4     "coordinator_stub_wsdl"];
    // No need to specify default_tx_provider here.
  };
};
```

The preceding configuration can be described as follows:

1. The `ws_coordination_service` plug-in is needed only on the client side. Artix does *not* support auto-loading of this plug-in; you must explicitly include it in the `orb_plugins` list.
The `ws_coordination_service` plug-in implicitly loads the `wsat_protocol` plug-in as well. Hence, it is unnecessary to include `wsat_protocol` plug-in in the `orb_plugins` list on the client side.
2. This line specifies that WS-AT is the default transaction provider. This implies that whenever a client initiates a transaction (for example, by calling `begin_transaction()`), Artix creates a new WS-AT transaction by default.
3. The server needs to load the `wsat_protocol` plug-in, in order to process incoming atomic transactions coordination contexts and to propagate transaction contexts. The `coordinator_stub_wsdl` plug-in enables the server to talk to the WS-Coordination service on the client side.
4. Strictly speaking, it is unnecessary to specify a default transaction provider on the server side. On the server side, the transaction provider is automatically determined by the incoming transaction context.
If the server needs to initiate its own transactions, however, it would be appropriate to set the default transaction provider here also.

References

The specifications for WS-AtomicTransactions and WS-Coordination are available at the following locations:

- [WS-AtomicTransactions](http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-AtomicTransaction.pdf)
(<http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-AtomicTransaction.pdf>).
- [WS-Coordination](http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-Coordination.pdf)
(<http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-Coordination.pdf>).

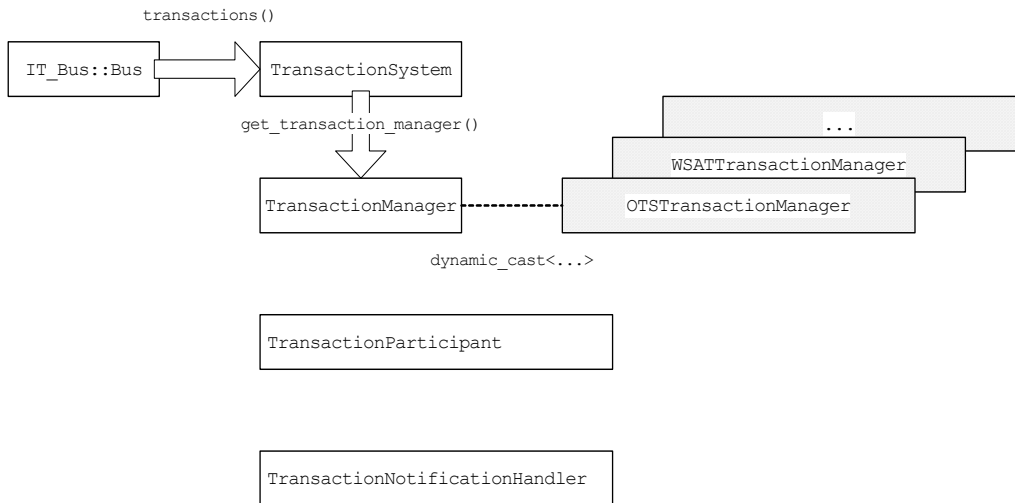
Transaction API

Overview

Figure 37 shows an overview of the main classes that make up the Artix transaction API. The Artix transaction API is designed to function as a generic wrapper for a wide variety of specific transaction systems. As long as your code is restricted to using the generic classes, you will be able to switch between any of the transaction systems supported by Artix.

On the server side it is likely that you will need to access advanced functionality, which is available only from technology-specific transaction manager classes, such as `OTSTransactionManager` or `WSATTransactionManager`.

Figure 37: Overview of the Artix Transaction API



Accessing the transaction system

To access the Artix transaction system, call the `transactions()` function on the `Bus`. The returned `IT_Bus::TransactionSystem` reference provides the starting point for accessing all aspects of Artix transactions.

The `IT_Bus::Bus::transactions()` function has the following signature:

```
IT_Bus::TransactionSystem&
transactions() IT_THROW_DECL((IT_Bus::Exception));
```

TransactionSystem class

The `IT_Bus::TransactionSystem` class provides the basic functions needed for transaction demarcation on the client side (`begin_transaction()`, `commit_transaction()` and `rollback_transaction()`). For more details see [“Transaction Demarcation” on page 539](#).

To access server-side functions and advanced client-side functions, you must call `IT_Bus::TransactionSystem::get_transaction_manager()` to obtain an `IT_Bus::TransactionManager` instance.

TransactionManager class

The `IT_Bus::TransactionManager` class provides server-side functions and advanced transaction functionality. For the server side, the most important member function is `IT_Bus::TransactionManager::enlist()`, which enables you to implement a transactional resource by enlisting a transaction participant object.

In order to support multiple transaction systems, the `TransactionManager` class is designed as a facade, which is layered above a specific implementation. In some cases, if the functionality provided by the generic `TransactionManager` is not sufficient, you might need to downcast the `TransactionManager` reference to one of the following types:

- [OTSTransactionManager class](#).
 - [WSATTransactionManager class](#).
-

OTSTransactionManager class

The `IT_Bus::OTSTransactionManager` class provides access to an underlying CORBA OTS implementation of the transaction system. Using this class, you can access the `CosTransactions::Coordinator` and the `CosTransactions::Current` objects for this transaction.

A discussion of the CORBA OTS is beyond the scope of this guide. For more details, see the *CORBA OTS Guide* (<http://www.iona.com/support/docs/orbix/6.2/develop.xml>), which is available from the Orbix documentation suite.

WSATTransactionManager class

The `IT_Bus::WSATTransactionManager` class provides access to an underlying WS-AT implementation of the transaction system. Currently, the `WSATTransactionManager` class provides access to the WS-AT context, which is included in a SOAP header with every transactional operation call.

TransactionParticipant base class

If you want to implement a transactional resource on the server side, you can define and implement a class that inherits from the `IT_Bus::TransactionParticipant` base class. The `TransactionParticipant` class receives callbacks from the transaction manager that are used to coordinate the commit or rollback steps with other transaction participants. For more details, see [“Participants and Resources” on page 542](#).

There are alternative ways of implementing a transactional resource, which do not require you to implement a `TransactionParticipant` class. Some transaction managers (for example, `OTSTransactionManager`) support alternative approaches.

TransactionNotificationHandler base class

If you want to synchronize certain actions with the committing or rolling back of a transaction, you can define and implement a class that inherits from the `IT_Bus::TransactionNotificationHandler` base class. The `IT_Bus::TransactionNotificationHandler` class receives notification callbacks from the transaction manager whenever a transaction is either committed or rolled back.

Transaction Demarcation

Overview

On the client side, the functions for beginning and committing (or rolling back) a transaction are collectively referred to as *transaction demarcation* functions. Within a given thread, any Artix operations invoked after the transaction *begin* and before the transaction *commit* (or *rollback*) are implicitly associated with the transaction. The transaction demarcation functions are typically the only functions that you need on the client side. For a detailed example of how to use the transaction demarcation functions, see “[Client Example](#)” on page 571.

TransactionSystem member functions

[Example 238](#) shows the public member functions of the `IT_Bus::TransactionSystem` class.

Example 238: The `IT_Bus::TransactionSystem` Class

```
// C++
namespace IT_Bus
{
    class IT_BUS_API TransactionSystem
        : public virtual RefCountedBase
    {
    public:
        virtual ~TransactionSystem();

        virtual void
        begin_transaction() IT_THROW_DECL((Exception)) = 0;

        virtual Boolean
        commit_transaction(
            Boolean report_heuristics
        ) IT_THROW_DECL((Exception)) = 0;

        virtual void
        rollback_transaction() IT_THROW_DECL((Exception)) = 0;

        virtual TransactionManager&
        get_transaction_manager(
            const String&
            tx_manager_type=DEFAULT_TRANSACTION_TYPE
        ) const = 0;
    };
};
```

Example 238:*The `IT_Bus::TransactionSystem` Class*

```

) IT_THROW_DECL((Exception)) = 0;

virtual Boolean
within_transaction() = 0;
...
// String constants for transaction manager types
static const String      DEFAULT_TRANSACTION_TYPE;
static const String      WSAT_TRANSACTION_TYPE;
static const String      OTS_TRANSACTION_TYPE;
static const String      XA_TRANSACTION_TYPE;
...
};

typedef Var<TransactionSystem> TransactionSystem_var;
typedef TransactionSystem* TransactionSystem_ptr;
};

```

Client transaction functions

The following functions are used to demarcate transactions on the client side:

- `begin_transaction()`—initiates a transaction on the client side. Implicitly, a new transaction is created and associated with the current thread.
- `commit_transaction()`—ends the transaction normally, making any changes permanent.
- `rollback_transaction()`—aborts the transaction, rolling back any changes.

Other transaction functions

In addition to the preceding demarcation functions, which are intended for use on the client side, the `TransactionSystem` class also provides the following functions, which can be used both on the client side and on the server side:

- `within_transaction()`—returns `true` if the current thread is associated with a transaction; otherwise, `false`.
- `get_transaction_manager()`—returns a reference to an `IT_Bus::TransactionManager` object, which provides access to advanced transaction features.

Typically, a `TransactionManager` object is needed on the server side in order to enlist participants in a transaction (for example, see [“Transaction Participants” on page 543](#)). For advanced applications, you can also downcast the `TransactionManager` reference to get a particular implementation of the transaction system (for example, an `IT_Bus::OTSTransactionManager` object or an `IT_Bus::WSATTransactionManager` object).

Participants and Resources

Overview

This section describes those aspects of server side programming which enable you to update a persistent resource transactionally.

In this section

This section contains the following subsections:

Transaction Participants	page 543
Interposition	page 550

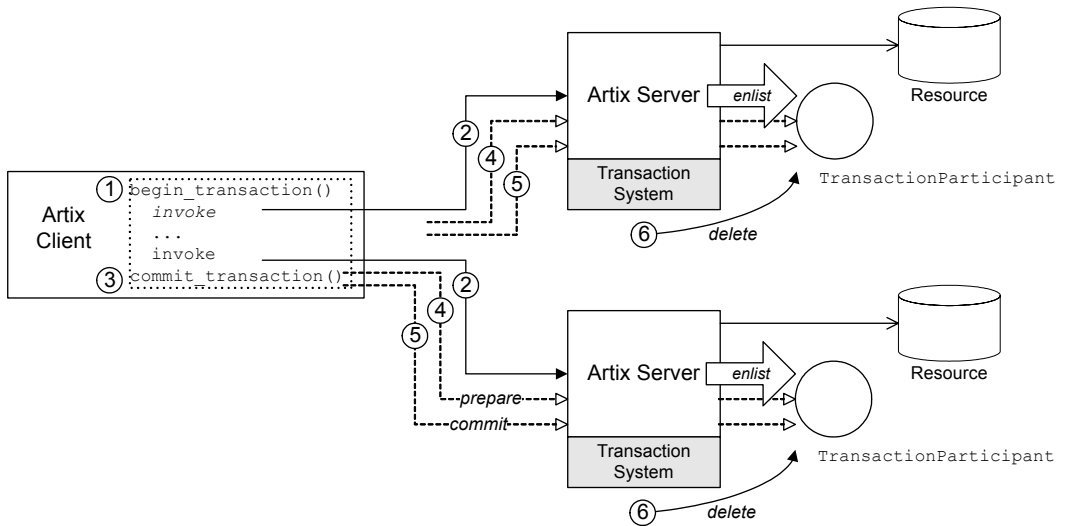
Transaction Participants

Overview

A transaction participant is an object on the server side that interfaces between the Artix transaction manager and a persistent resource. The role of the transaction participant is to receive callbacks from the transaction manager, which tell the participant whether to make pending changes permanent or whether to abort the current transaction and return the resource to its previous consistent state.

Figure 38 shows an example of a two-phase commit involving two transaction participant instances. Any operations meant to be transactional should start by creating a transaction participant object and enlisting it with the transaction manager.

Figure 38: *Transaction Participants in a 2-Phase Commit Protocol*



Participants in a 2-phase commit

As shown in [Figure 38](#), the transaction participants participate in a two-phase commit as follows:

Stage	Description
1	The client calls <code>begin_transaction()</code> to initiate a distributed transaction.
2	Within the transaction, the client calls transactional operations on Server A and on Server B. In order to participate in the distributed transaction, the servant code creates a new transaction participant and enlists it with the transaction manager.
3	The client calls <code>commit_transaction()</code> to make permanent any changes caused during the transaction.
4	The transaction system performs the prepare phase by polling all of the remote transaction participants (the first phase of a two-phase commit). On the server side, the transaction manager calls <code>prepare()</code> on all of the transaction participants.
5	The transaction system performs the commit or rollback phase by sending a notification to all of the remote transaction participants (the second phase of a two-phase commit). On the server side, the transaction manager calls <code>commit()</code> or <code>rollback()</code> on all of the transaction participants.
6	When the transaction is finished, the transaction manager automatically deletes the associated transaction participant instances.

Implementing a transaction participant

To implement a transaction participant, define a class that inherits from the `IT_Bus::TransactionParticipant` base class and implement all of its member functions.

Alternatives to the Artix transaction participant

Implementing and enlisting an Artix `TransactionParticipant` class is not the only way to make a WSDL operation transactional. By drilling down to the underlying transaction manager type (for example, `IT_Bus::OTSTransactionManager`) it is sometimes possible to use an alternative API supported by a specific transaction system.

For example, the following demonstration shows how to use the OTS transaction system:

```
ArtixInstallDir/artix/Version/demos/transactions/orbix_client_art
ix_server
```

Enlisting a transaction participant

[Example 239](#) shows an example of how to enlist a participant instance in a transaction. You must enlist a participant at the start of any transactional WSDL operation. [Example 239](#) shows a sample implementation of a WSDL operation, `transactional_op()`, which is called in the context of a transaction.

Example 239: Example of Enlisting a Transactional Participant

```
// C++
void
HelloWorldServantImpl::transactional_op(
    const IT_Bus::String value
) IT_THROW_DECL((IT_Bus::Exception))
{
    cout << "HelloWorld transactional_op() called" << endl;

1   IT_Bus::Bus_var bus = this->get_bus();
2   if (bus->transactions().within_transaction())
    {
        cout << "This is a transaction" << endl;

3       TXParticipant * participant = new TXParticipant(this);
4       bus->transactions().get_transaction_manager().enlist(
            participant,
            true
        );

5       // Implementation of 'transactional_op()' comes here.
        // Includes writing to DB or other persistent resources.
        // (not shown)
        ...
    }
}
```

Example 239: *Example of Enlisting a Transactional Participant*

```

else
{
    cout << "No transaction" << endl;
    IT_Bus::Exception ex("Invocation not in transaction");
    throw ex;
}
}

```

The preceding code example can be explained as follows:

1. The `get_bus()` function is a standard servant function that returns a stored reference to the Bus instance.
2. In this example, the `transactional_op()` operation *requires* a transaction. If it is not called in the context of a transaction, it raises an exception back to the client.

It is an implementation decision whether or not an operation should require a transaction. In some cases, it may be appropriate for the operation to proceed with or without a transaction.

3. The `TXParticipant` class is a sample participant class, which is implemented by inheriting from `IT_Bus::TransactionParticipant`. In this example, a new `TXParticipant` instance is created every time `transactional_op()` is called.
4. This line enlists the participant in the transaction, ensuring that the participant receives callbacks either to commit or rollback any changes.

The second parameter is a boolean flag that specifies the kind of participant:

- ◆ `true` indicates a *durable participant*, which participates in all phases of the transaction.
- ◆ `false` indicates a *volatile participant*, which is only guaranteed to participate in the prepare phase of the 2PC protocol. There is no guarantee that a volatile participant will participate in the commit phase.

- The implementation of `transactional_op()` involves writing to a persistent resource. The committing or rolling back of any changes to this persistent resource is controlled by the enlisted `TXPersistent` instance.

TransactionParticipant member functions

[Example 240](#) shows the public member functions of the `IT_Bus::TransactionParticipant` class.

Example 240:*The `IT_Bus::TransactionParticipant` Class*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API TransactionParticipant
        : public virtual RefCountedBase
    {
    public:
        virtual ~TransactionParticipant();

        enum VoteOutcome {
            VoteCommit,
            VoteRollback,
            VoteReadOnly
        };

        // 1PC Functions.
        virtual void commit_one_phase()=0;

        // 2PC Functions.
        virtual VoteOutcome prepare()=0;
        virtual void        commit()=0;
        virtual void        rollback()=0;

        // Getting the transaction manager.
        virtual String
        preferred_transaction_manager()=0;

        virtual void
        set_manager(
            TransactionManager* tx_manager
        )=0;
        ...
    };
    typedef Var<TransactionParticipant>
TransactionParticipant_var;
}
```

Example 240:*The IT_Bus::TransactionParticipant Class*

```
typedef TransactionParticipant* TransactionParticipant_ptr;  
};
```

1PC callback function

The following function is called during a one-phase commit:

- `commit_one_phase()`—the implementation of this function should make permanent any changes associated with the current transaction.

2PC callback functions

The following functions are called during a two-phase commit:

- `prepare()`—called during *phase one* of a two-phase commit. Before returning, this function should write a recovery log to persistent storage. The recovery log should contain whatever data would be necessary to restore the system to a consistent state, in the event that the server crashes before the transaction is finished.

The `prepare()` function also votes on whether to commit or roll back the transaction overall, by returning one of the following vote outcomes:

- ◆ `IT_Bus::TransactionParticipant::VoteCommit`—vote to commit the transaction.
- ◆ `IT_Bus::TransactionParticipant::VoteRollback`—vote to roll back the transaction. For example, you would return `VoteRollback`, if an error occurred while attempting to write the recovery log.
- ◆ `IT_Bus::TransactionParticipant::VoteReadOnly`—explicitly request *not* to be included in the commit phase of the 2PC protocol.
- `commit()`—called during *phase two* of a two-phase commit, if the transaction outcome was successful overall. The implementation of this function should make permanent any changes associated with the current transaction.
- `rollback()`—called during *phase two* of a two-phase commit, if the transaction must be aborted. The implementation of this function should undo any changes associated with the current transaction, returning the system to the state it was in before.

Getting the transaction manager

After the transaction participant is enlisted by a transaction manager instance, the transaction system calls back to pass a transaction manager to the participant. The following functions are relevant to this callback behavior:

- `preferred_transaction_manager()`—called just after the participant is enlisted. The return value is a string that tells the transaction system what type of transaction manager the participant requires. The following return strings are supported:
 - ◆ `DEFAULT_TRANSACTION_TYPE`—no preference; use the current default.
 - ◆ `OTS_TRANSACTION_TYPE`—prefer the `IT_Bus::OTSTransactionManager` interface (manager for CORBA OTS transactions).
 - ◆ `WSAT_TRANSACTION_TYPE`—prefer the `IT_Bus::WSATTransactionManager` interface (manager for WS-AtomicTransactions).
- `set_manager()`—called after the `preferred_transaction_manager()` call. The transaction system calls `set_manager()` to pass a transaction manager of the preferred type to the participant. If the type of transaction manager requested by the participant differs from the one currently in use, Artix uses *interposition* to simulate the preferred transaction manager type.

For more details about interposition, see [“Interposition” on page 550](#).

Interposition

What is interposition?

Sometimes, there can be a mismatch between the transaction API used by the application code and the type of the underlying transaction system. For example, imagine that you have a legacy CORBA server that manages transactions with CORBA OTS. If you migrate this server code to a WS-AT-based Artix service, you would obtain a mismatch between the transaction API used by the application code (which is CORBA OTS-based) and the underlying transaction system (which is WS-AT).

To bridge this API mismatch, Artix uses *interposition*. With interposition, the Artix runtime provides the application code with an object of the preferred type (for example, an `OTSTransactionManager` object), but the object is merely a facade, whose calls are ultimately translated into a form suitable for the underlying transaction system (for example, WS-AT).

Interposition matrix

Artix supports interposition between *every* permutation of transaction systems. Internally, Artix converts calls made on a specific transaction API into a technology-neutral API. The calls are then converted from the technology-neutral API into one of the supported transaction APIs.

Using interposition

As an example of interposition, consider a service that loads the WS-AT transaction system (for example, see [“Configuring WS-AT” on page 533](#)), but actually implements the transaction functionality using the CORBA OTS programming interface. In this case, it is necessary for the `TransactionParticipant` implementation to request explicitly an OTS transaction manager, instead of the default WS-AT transaction manager.

[Example 241](#) shows the implementation of the `preferred_transaction_manager()` function and the `set_manager()` function for the transaction participant implementation, `TxParticipant`.

Example 241: *Example of a `TransactionParticipant` that Uses Interposition*

```
// C++
...
IT_Bus::String
TxParticipant::preferred_transaction_manager()
{
```

Example 241: *Example of a TransactionParticipant that Uses Interposition*

```
    return IT_Bus::TransactionSystem::OTS_TRANSACTION_TYPE;
}

void
TXParticipant::set_manager(
    IT_Bus::TransactionManager* tx_manager
)
{
    m_ots_tx_manager =
        dynamic_cast<IT_Bus::OTSTransactionManager*>(tx_manager);
}
```

When Artix calls back on `set_manager()`, it passes a transaction manager object, `tx_manager`, of `OTSTransactionManager` type. There is no need to query the type of the `tx_manager` object before downcasting it, because its type is already specified by the `preferred_transaction_manager()` callback.

Threading

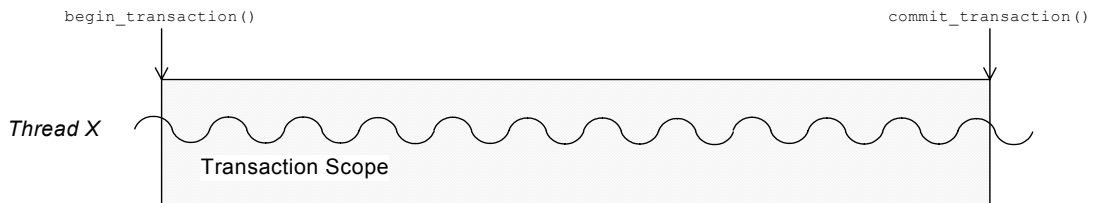
Overview

Artix supports a threading API that enables you to change the thread affinity of a given transaction. Using the `attach_thread()` and `detach_thread()` functions, you can flexibly re-assign threads to a transaction (subject to the limitations imposed by the underlying transaction system).

Default client threading model

Figure 39 shows the default threading model for transaction on the client side. When you call `begin_transaction()`, Artix creates a new transaction and attaches it to the current thread. So long as the transaction remains attached, any WSDL operations called from the current thread become part of the transaction. When you call `commit_transaction()` (or `rollback_transaction()`, if the transaction must be aborted), the transaction is deleted.

Figure 39: *Default Client Threading Model*



Transaction identifiers

A *transaction identifier* is an opaque identifier of type `IT_Bus::TransactionIdentifier` that identifies a transaction uniquely. Depending on the underlying transaction system, a transaction identifier can be downcast (using `dynamic_cast<...>`) to an implementation-specific transaction identifier.

For example, if OTS is the underlying transaction system, the transaction identifier can be downcast to an instance of an `OTSTransactionIdentifier`. The OTS transaction identifier provides access to implementation-specific features, such as the `CosTransaction::Control` class.

Controlling thread affinity

On the client side, thread affinity is controlled by the following `TransactionManager` member functions:

Example 242: Functions for Controlling Thread Affinity

```
// C++
namespace IT_Bus
{
    class IT_BUS_API TransactionManager
        : public virtual RefCountedBase
    {
    public:
        virtual TransactionIdentifier* detach_thread()=0;

        virtual Boolean          attach_thread(
            TransactionIdentifier* tx_identifier
        ) = 0;

        virtual TransactionIdentifier* get_tx_identifier()=0;
        ...
    };
};
```

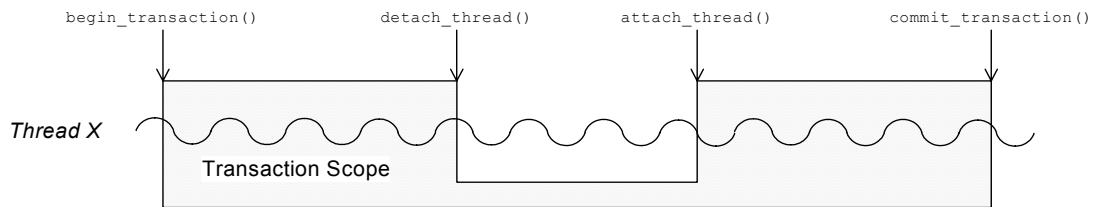
These functions can be explained as follows:

- `detach_thread()`
Detach the transaction from the current thread. After the call to `detach_thread()`, WSDL operations called from the current thread do not participate in the transaction. The returned transaction identifier can be used to re-attach the transaction to the current thread at a later stage.
- `attach_thread()`
Attach the transaction, specified by the `tx_identifier` argument, to the current thread.
- `get_tx_identifier()`
Return the identifier of the transaction that is attached to the current thread. If no transaction is attached, return `NULL`.

Detaching and re-attaching a transaction to a thread

Figure 40 shows how to use the `detach_thread()` and `attach_thread()` functions to suspend temporarily the association between a transaction and a thread. This can be useful if, in the midst of a transaction, you need to perform some non-transactional tasks.

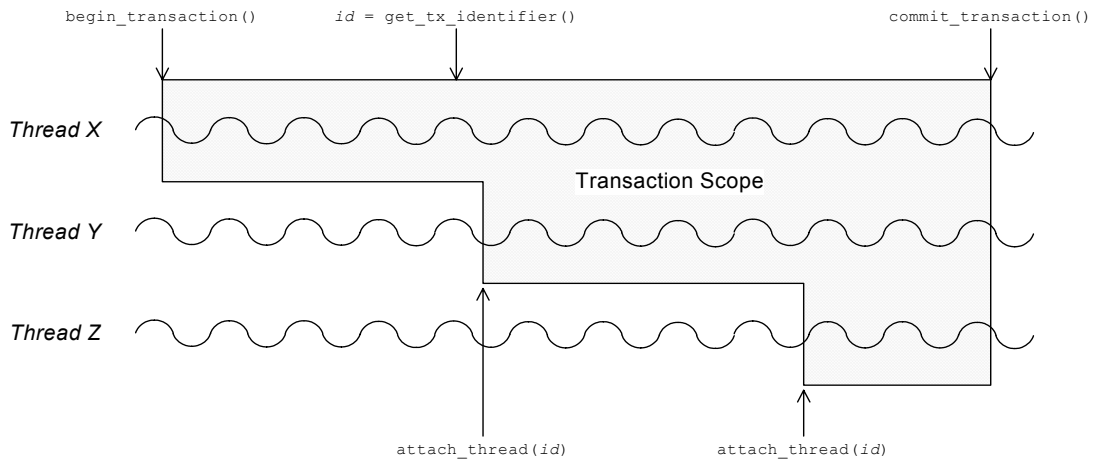
Figure 40: *Detaching and Re-Attaching a Transaction to a Thread*



Attaching a transaction to multiple threads

Figure 41 shows how to use the `get_tx_identifier()` and `attach_thread()` functions to associate a transaction with multiple threads. The `get_tx_identifier()` function is called from within the thread that initiated the transaction. The transaction ID can then be passed to the other threads, Y and Z, enabling them to attach the transaction.

Figure 41: *Attaching a Transaction to Multiple Threads*

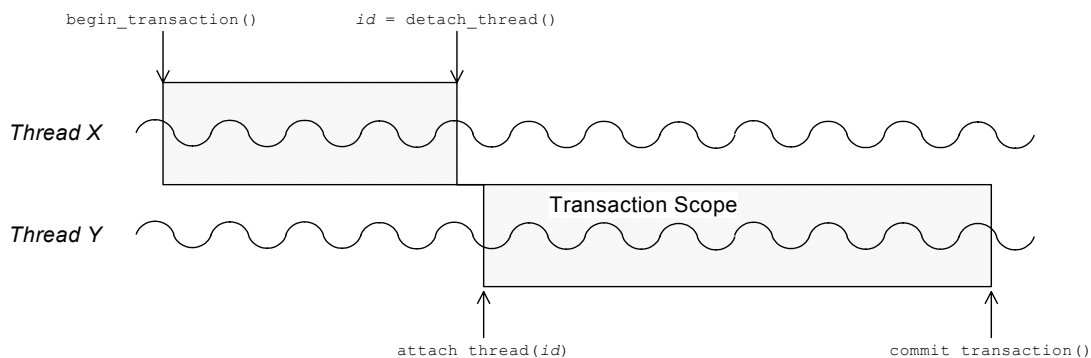


Note: Some transaction systems do not allow you to associate multiple threads with a transaction. In this case, an `attach_thread()` call fails (returning `false`), if you attempt to attach a second thread to the transaction.

Transferring a transaction from one thread to another

Figure 42 shows how to use the `detach_thread()` and `attach_thread()` functions to transfer a transaction from thread X to thread Y. The transaction ID returned from the `detach_thread()` call must be passed to thread Y, enabling it to attach the transaction.

Figure 42: *Transferring a Transaction from One Thread to Another*



Note: Some transaction systems do not allow you to transfer a transaction from one thread to another. In this case, an `attach_thread()` call fails (returning `false`), unless you are re-attaching the original thread to the transaction.

Transaction Propagation

Overview

In a multi-tier application, Artix automatically propagates transactions from tier to tier. This ensures that all of the processes that are relevant to the outcome of a transaction can participate in the transaction. You do not have to do anything special to switch on transaction propagation; it is enabled by default. However, the receiver of a transaction context must have a transaction plug-in loaded, otherwise the transaction context would be ignored.

Transaction contexts

A transaction context is a data structure that is transmitted to a remote server and used to recreate the transaction at a remote location. The type of transaction context that is transmitted depends on the middleware protocol. Artix supports the following kinds of transaction context:

- *OTS transaction context*—a transaction context that is sent in a GIOP header (part of the CORBA standard).
 - *WS-AT transaction context*—a transaction context that is embedded in a SOAP header.
-

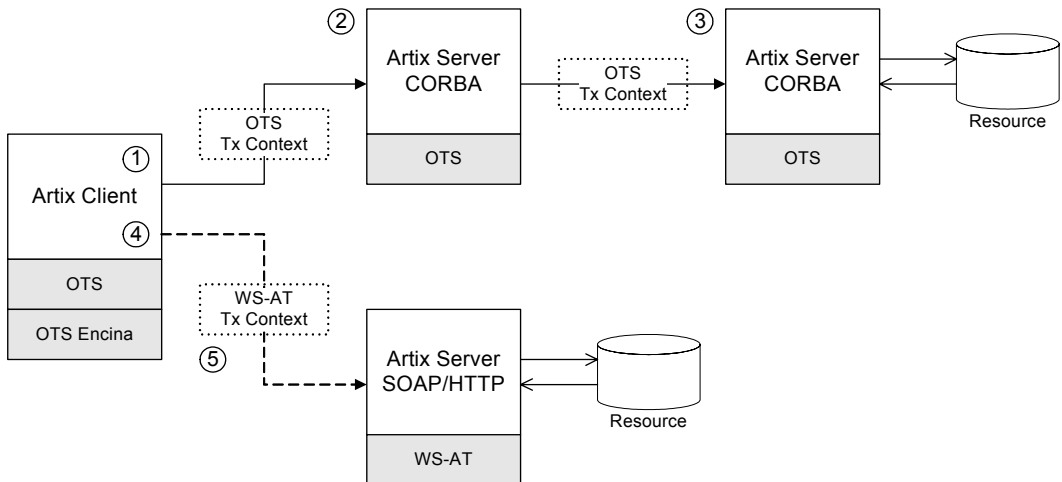
Propagation scenario

The propagation scenario shown in [Figure 43](#) shows two different kinds of transaction propagation, as follows:

- *Transaction propagation within a single middleware technology*—the OTS transaction context, which propagates across the top half of [Figure 43](#), illustrates a simple kind of propagation, where the client and the servers all use the same CORBA OTS transaction technology.
- *Transaction propagation across middleware technologies*—the WS-AT transaction context, which propagates across the bottom half of [Figure 43](#), illustrates a kind of propagation, where the transaction crosses technology domains. While the client uses OTS Encina to

manage the transaction, it must generate a WS-AT transaction context to send to the server. The ability to transform transaction contexts is known as *interposition*.

Figure 43: Overview of Different Kinds of Transaction Propagation



Scenario steps

The propagation scenario shown in [Figure 43](#) can be described as follows:

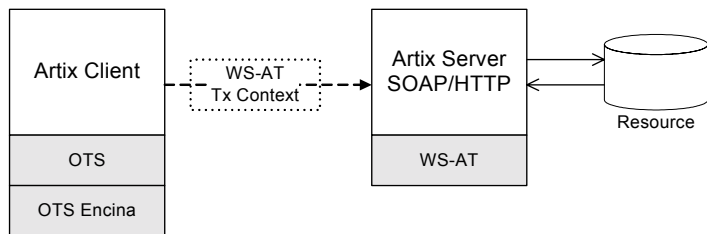
Stage	Description
1	The Artix client (which is configured to use the OTS Encina transaction system) initiates a transaction by calling the <code>begin_transaction()</code> function. The client then invokes a remote operation, which results in a request message being sent over an IIOp connection.
2	The request received by the server includes an OTS transaction context embedded in a GIOP header. Although this server does not participate directly in the transaction (it registers no resources), it is capable of propagating the transaction context to the next tier in the application.

Stage	Description
3	The third tier of the application receives a request containing an OTS transaction context. This server participates in the transaction by registering a database resource with the OTS transaction manager.
4	The client invokes a remote operation, which results in a request message being sent over a SOAP/HTTP connection.
5	In this case, Artix automatically translates the OTS transaction into a WS-AT transaction context, which is suitable for transmission in the header of the SOAP/HTTP request. There is no need to perform any special configuration or programming to enable interposition; it occurs automatically.

Limitation of using OTS Lite with propagation

Figure 44 shows an interposition scenario where the client, which uses an OTS transaction system, connects to a SOAP/HTTP server, which uses the WS-AT transaction system.

Figure 44: *Limitation of Transaction Propagation Using OTS Lite*



Because there is only one explicitly registered resource in this scenario (the database connected to the server), it would seem that the client could use an OTS Lite transaction manager for this scenario. In reality, however, the client *must* use the OTS Encina transaction manager. The reason for this is that Artix implicitly registers an interposition resource to bridge the OTS-to-WS-AT middleware boundary. Therefore, there are really two resources in this scenario.

In summary, interposition requires additional resources as follows:

- *OTS-to-WS-AT middleware boundary*—one interposition resource is registered automatically. Applications with one explicitly registered resource must use OTS Encina.
- *WS-AT-to-OTS middleware boundary*—no interposition resource required. Applications with one explicitly registered resource may use OTS Lite.

Suppressing propagation

Once you have selected a transaction system (for example, the application loads an OTS plug-in or a WS-AT plug-in), transaction contexts are propagated by default.

It is possible, however, to suppress transaction propagation selectively using the `IT_Bus::TransactionManager::detach_thread()` and `IT_Bus::TransactionManager::attach_thread()` functions. After calling `detach_thread()`, subsequent operation invocations do not participate in the transaction and, therefore, do not propagate any transaction context. You can re-establish an association with a transaction by calling `attach_thread()`.

For more details on these functions, see [“Threading” on page 552](#).

Notification Handlers

Overview

A *notification handler* is an object that can be used either on the server side or on the client side to record the outcome of a transaction. For example, you might use a notification handler to log transaction outcomes or to synchronize other events with a transaction.

Implementing a notification handler

To implement a notification handler, define a class that inherits from the `IT_Bus::TransactionNotificationHandler` base class and implement all of its member functions.

Enlisting a notification handler

To use a notification handler, you must enlist it with a `TransactionManager` object while there is a current transaction. You can enlist a notification handler at any time prior to the termination of the transaction.

[Example 243](#) shows how to enlist a sample notification handler, `NotificationHandlerImpl`.

Example 243: Example of Enlisting a Notification Handler

```
// C++
IT_Bus::Bus_var bus = ... // Get reference to Bus object
if (bus->transactions().within_transaction())
{
    // Enlist notification handler
    NotificationHandlerImpl * handler
        = new NotificationHandlerImpl();
    TransactionManager& tx_manager
        = bus->transactions().get_transaction_manager()
    tx_manager.enlist_for_notification(handler);
}
else
{
    IT_Bus::Exception ex("Invocation not in transaction");
    throw ex;
}
```


TransactionNotificationHandler member functions

[Example 244](#) shows the public member functions of the `IT_Bus::TransactionNotificationHandler` class. These operations will be called, only if an appropriate notification mechanism is available in the underlying transaction system.

Example 244: The `IT_Bus::TransactionNotificationHandler` Class

```
// C++
namespace IT_Bus
{
    class IT_BUS_API TransactionNotificationHandler
        : public virtual RefCountedBase
    {
    public:
        virtual ~TransactionNotificationHandler();

        virtual void commit_initiated(
            TransactionIdentifier_ptr tx_identifier
        )=0;
        virtual void committed()=0;
        virtual void aborted()=0;
        ...
    };

    typedef Var<TransactionNotificationHandler>
        TransactionNotificationHandler_var;
    typedef TransactionNotificationHandler*
        TransactionNotificationHandler_ptr;
};
```

Notification callback functions

The following notification handler functions receive callbacks from the transaction manager:

- `commit_initiated()`—informs the handler that a commit has been initiated. This function is called before any participants are prepared.

Note: WS-AT does not support this notification point.

- `committed()`—informs the handler that the transaction completed successfully.
- `aborted()`—informs the handler that the transaction did not complete successfully and was aborted.

Reliable Messaging with MQ Transactions

Overview

This section describes how to enable reliable messaging with MQ transactions in your Artix applications. MQ transactions differ in several important respects from ordinary Artix transactions, in particular:

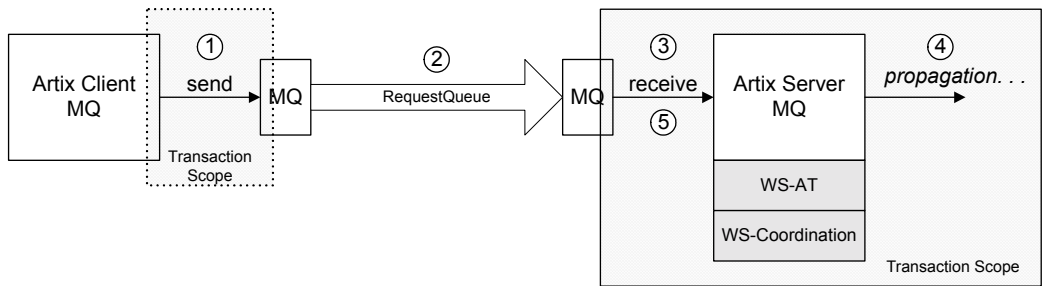
- MQ transactions are managed by a transaction manager that is internal to the MQ-Series product.
- MQ transactions are enabled by setting the relevant attributes of a WSDL port in the WSDL contract.
- You can *not* initiate and terminate MQ transactions on the client side using the Artix transaction API (for example, the functions in `IT_Bus::TransactionSystem` are not used for MQ on the client side).

On the client side, MQ transactions follow a completely different model from Artix transactions. On the server side, however, the MQ transaction is integrated with an Artix transaction, so that an incoming message is considered to have been processed, only if the Artix transaction completes successfully on the server side.

Oneway invocation scenario

Figure 45 shows a oneway invocation scenario, where an Artix client invokes oneway operations on an Artix server over the MQ transport with MQ transactions enabled. Because the WSDL operations are *oneway* (that is, consisting only of output messages), the MQ transport does not require a reply queue in this scenario.

Figure 45: *Oneway Operation Invoked Over an MQ Transport with MQ Transactions Enabled*



Description of oneway invocation

The oneway operation invocation shown in Figure 45 is executed in the following stages:

Stage	Description
1	When the client invokes a oneway operation over MQ, an MQ transaction is initiated. After the request message is pushed onto the client side of the MQ request queue, the MQ transaction is committed. Note: The client MQ transaction is local and does <i>not</i> extend beyond the client side.
2	MQ-Series is responsible for reliably transmitting the request message from the client side of the MQ transport to the server side of the MQ transport.
3	When the server pulls the request message off the incoming queue, an Artix transaction is initiated before dispatching the request to the relevant Artix servant.

Stage	Description
4	If the Artix servant now invokes operations on some other Artix servers, these invocations occur within a transaction context. Hence, these follow-on invocations propagate a transaction context (for example, a WS-AT context) and enable the remote servers to participate in the transaction.
5	If the operation completes its work successfully, the transaction is committed and the request message permanently disappears from the queue. On the other hand, if the operation is unsuccessful, the transaction is rolled back and the request message is pushed <i>back</i> onto the queue. The request message is immediately reprocessed (the maximum number of times the message can be processed is determined by the queue's backout threshold—see “Configuring the backout threshold” on page 569).

Oneway client configuration

To enable transactional semantics for a client that invokes oneway operations over the MQ transport, you should define a WSDL port as shown in [Example 245](#).

Example 245: WSDL Port Configuration for Oneway Client Over MQ

```
<wsdl:service name="MQService">
  <wsdl:port binding="tns:BindingName" name="PortName">
    <mq:client QueueManager="MY_DEF_QM"
              QueueName="HW_REQUEST"

              AccessMode="send"
              CorrelationStyle="correlationId"
              Transactional="internal"
              Delivery="persistent"
              UsageStyle="peer"

    />
    ...
  </wsdl:port>
</wsdl:service>
```

Because the invocation is oneway, there is no need to specify a reply queue manager. To enable transactions, you must set the `Transactional` attribute to `internal` and the `Delivery` attribute to `persistent`.

Oneway server configuration

On the server side, you must configure both the WSDL contract and the Artix configuration file appropriately for using MQ transactions.

WSDL Contract Configuration

To enable transactional semantics for a server that receives oneway invocations over the MQ transport, you should define a WSDL port as shown in [Example 246](#).

Example 246: WSDL Port Configuration for Oneway Server Over MQ

```
<wsdl:service name="MQService">
  <wsdl:port binding="tns:BindingName" name="PortName">
    ...
    <mq:server QueueManager="MY_DEF_QM"
              QueueName="HW_REQUEST"

              AccessMode="receive"
              CorrelationStyle="correlationId"
              Transactional="internal"
              Delivery="persistent"
              UsageStyle="peer"

    />
  </wsdl:port>
</wsdl:service>
```

To enable transactions, you must set the `Transactional` attribute to `internal` and the `Delivery` attribute to `persistent`.

Artix Configuration File

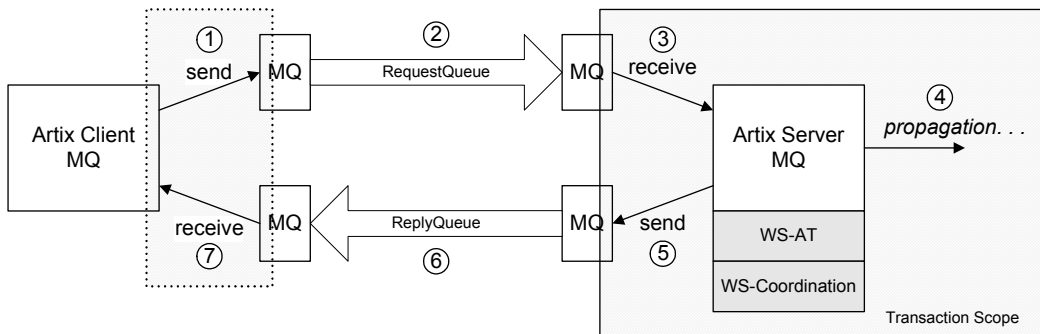
On the server side, Artix initiates a transaction whenever it receives a request message from the MQ transport. Because this transaction is managed by an Artix transaction manager, you must load and configure one of the Artix transaction systems (for example, OTS or WS-AT).

For details of how to select a transaction system, see [“Selecting the Transaction System”](#) on page 525.

Synchronous invocation scenario

Figure 46 shows a synchronous invocation scenario, where an Artix client invokes normal operations on an Artix server over the MQ transport with MQ transactions enabled. Because the WSDL operations are *synchronous* (that is, consisting of output messages and input messages), the MQ transport requires a reply queue.

Figure 46: Synchronous Operation Invoked Over the MQ Transport with MQ Transactions Enabled



Description of synchronous invocation

The synchronous operation invocation shown in Figure 46 is executed in the following stages:

Stage	Description
1	When the client invokes a synchronous operation over MQ, an MQ transaction is initiated.
2	MQ-Series is responsible for reliably transmitting the request message from the client side of the MQ transport to the server side of the MQ transport.
3	When the server pulls the request message off the incoming queue, an Artix transaction is initiated before dispatching the request to the relevant Artix servant.

Stage	Description
4	If the Artix servant now invokes operations on some other Artix servers, these invocations occur within a transaction context. Hence, these follow-on invocations propagate a transaction context (for example, a WS-AT context) and enable the remote servers to participate in the transaction.
5	If the operation completes its work successfully, the transaction is committed, the request message permanently disappears from the request queue, and a reply message gets pushed onto the reply queue. On the other hand, if the operation is unsuccessful, the transaction is rolled back. No reply message is sent and the request message is pushed <i>back</i> onto the request queue. The request message is immediately reprocessed (the maximum number of times the message can be processed is determined by the request queue's backout threshold—see “Configuring the backout threshold” on page 569).
6	MQ-Series is responsible for reliably transmitting the reply message from the server side of the MQ transport to the client side of the MQ transport.
7	When the client receives the reply message, the synchronous operation call returns and the client transaction is committed. Because the client is independent of the server side transaction, however, it is not possible for the client code to receive a rollback exception from the server. It is possible to manage blocked calls by defining the <code>Timeout</code> attribute on the <code>mq:client</code> element in the WSDL contract. If the timeout is exceeded, an exception will be thrown.

Synchronous client configuration

To enable transactional semantics for a client that invokes synchronous operations over the MQ transport, you should define a WSDL port as shown in [Example 247](#).

Example 247: WSDL Port Configuration for Synchronous Client Over MQ

```
<wsdl:service name="MQService">
  <wsdl:port binding="tns:BindingName" name="PortName">
    <mq:client QueueManager="MY_DEF_QM"
```

Example 247: WSDL Port Configuration for Synchronous Client Over MQ

```

        QueueName="HW_REQUEST"
        ReplyQueueManager="MY_DEF_QM"
        ReplyQueueName="HW_REPLY"
        AccessMode="send"
        CorrelationStyle="correlationId"
        Transactional="internal"
        Delivery="persistent"
        UsageStyle="responder"
    />
    ...
</wsdl:port>
</wsdl:service>

```

To enable transactions, you must set the `Transactional` attribute to `internal` and the `Delivery` attribute to `persistent`.

Synchronous server configuration

On the server side, you must configure both the WSDL contract and the Artix configuration file appropriately for using MQ transactions.

WSDL Contract Configuration

To enable transactional semantics for a server that receives synchronous invocations over the MQ transport, define a WSDL port as shown in [Example 248](#).

Example 248: WSDL Port Configuration for Synchronous Server Over MQ

```

<wsdl:service name="MQService">
  <wsdl:port binding="tns:BindingName" name="PortName">
    ...
    <mq:server QueueManager="MY_DEF_QM"
      QueueName="HW_REQUEST"
      ReplyQueueManager="MY_DEF_QM"
      ReplyQueueName="HW_REPLY"
      AccessMode="receive"
      CorrelationStyle="correlationId"
      Transactional="internal"
      Delivery="persistent"
      UsageStyle="responder"
    />
  </wsdl:port>
</wsdl:service>

```


To enable transactions, you must set the `Transactional` attribute to `internal` and the `Delivery` attribute to `persistent`.

Artix Configuration File

On the server side, Artix initiates a transaction whenever it receives a request message from the MQ transport. Because this transaction is managed by an Artix transaction manager, you must load and configure one of the Artix transaction systems (for example, OTS or WS-AT).

For details of how to select a transaction system, see [“Selecting the Transaction System” on page 525](#).

Configuring the backout threshold

You can configure the backout threshold using the `runmqsc` command-line tool, which is provided as part of the *MQ-Series* product. To configure a queue to use backouts, set the following MQ attributes:

- `BOTHRESH`—the backout threshold, which defines the maximum number of times a message can be pushed back onto the queue.
- `BOQNAME`—the backout queue name. If the current backout count equals the backout threshold, Artix puts the message onto the backout queue whose name is given by `BOQNAME`.

Hence, the `BOQNAME` queue would contain all of the messages that have been rolled back more than `BOTHRESH` times. The administrator can then manually examine the messages stored in the `BOQNAME` queue and take appropriate remedial action.

For more details about how to set MQ attributes, see your *MQ-Series* user documentation.

Accessing the backout count

On the server side, you can obtain the backout count for the current message using Artix contexts. To access the current backout count, perform the following steps:

1. Retrieve the server context identified by the `IT_ContextAttributes::MQ_INCOMING_MESSAGE_ATTRIBUTES` `QName`.
2. Cast the returned context instance to the `IT_ContextAttributes::MQMessageAttributesType` type.
3. Invoke the `getBackoutCount()` function to access the current backout count.

For more details about programming with Artix contexts, see [“Artix Contexts” on page 179](#).

Client Example

Overview

This section describes a transactional Artix client that connects to two remote transactional Artix servers. The client uses the Artix transaction demarcation API to delimit the transaction. The client must also be configured to load a transaction system plug-in (see [“Selecting the Transaction System” on page 525](#)).

WSDL sample

[Example 249](#) is a sample WSDL contract that defines the `Data` port type with two operations, `read()` and `write()`. The effect of these operations is to read or write a single integer value from persistent storage. The `write()` operation is required to be transactional (but this does not need to be indicated in the WSDL contract).

Example 249: Sample WSDL Contract for the Data Port Type

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="data.idl"
  targetNamespace="http://schemas.iona.com/idl/data.idl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  ...
  xmlns:tns="http://schemas.iona.com/idl/data.idl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://schemas.iona.com/idltypes/data.idl">
  <types>
    <schema
      targetNamespace="http://schemas.iona.com/idltypes/data.idl"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <element name="Data.read.value" type="xsd:int"/>
      <element name="Data.write.value" type="xsd:int"/>
    </schema>
  </types>
  <message name="Data.read"/>
  <message name="Data.readResponse">
    <part element="xsd1:Data.read.value" name="value"/>
  </message>
  <message name="Data.write">
    <part element="xsd1:Data.write.value" name="value"/>
  </message>
  <message name="Data.writeResponse"/>
```

Example 249: *Sample WSDL Contract for the Data Port Type*

```

<portType name="Data">
  <operation name="read">
    <input message="tns:Data.read" name="read"/>
    <output message="tns:Data.readResponse"
      name="readResponse"/>
  </operation>
  <operation name="write">
    <input message="tns:Data.write" name="write"/>
    <output message="tns:Data.writeResponse"
      name="writeResponse"/>
  </operation>
</portType>
...
</definitions>

```

Client example

Example 250 shows how to use the transaction demarcation functions in an Artix client. Two remote services, `DataServiceA` and `DataServiceB`, participate in the transaction. Hence, this example requires support for the two-phase commit protocol.

Example 250: *Transaction Demarcation in an Artix Client*

```

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

#include "DataClient.h"

IT_USING_NAMESPACE_STD

using namespace ArtixTransactions;
using namespace IT_Bus;

int
main(
  int argc,
  char* argv[]
)
{
  IT_Bus::Bus_var bus;
  bus = IT_Bus::init(argc, argv);

```

Example 250: *Transaction Demarcation in an Artix Client*

```

1      try
      {
          // Client of "DataServiceA" WSDL service.
          DataClient clientA (serviceA_wsdl, serviceA_name, bus);

          // Client of "DataServiceB" WSDL service.
          DataClient clientB (serviceB_wsdl, serviceB_name, bus);

          IT_Bus::Int valueA, valueB;

2      bus->transactions().begin_transaction();

          // Perform a 2PC transaction
          clientA.read(valueA);
          clientB.read(valueB);
          cout << "The current values are:" << endl;
          cout << "\tA value: " << valueA << endl;
          cout << "\tB value: " << valueB << endl;

          cout << "Changing both values" << endl;
          clientA.write(valueA + 10);
          clientB.write(valueB - 10);

3      bus->transactions().commit_transaction(true);
      }
      catch(IT_Bus::Exception& e)
      {
4          if (bus->transactions().within_transaction() )
          {
5              cout << endl << "Aborting transaction!" << endl;
              bus->transactions().rollback_transaction();
          }
          return -1;
      }
      if (bus.get() != 0) { bus->shutdown(true); }
      return 0;
  }

```

The preceding code example can be explained as follows:

1. You should always enclose a transaction in a `try` block, because it might be necessary to catch an exception and roll back the transaction.
2. The `IT_Bus::TransactionSystem::begin_transaction()` call initiates the transaction.
3. The `IT_Bus::TransactionSystem::commit_transaction()` call attempts to commit the changes made to server A and server B. The boolean argument is the `report_heuristics` flag, which can take the following values:
 - ◆ `true`—specifies that heuristic decisions should be reported during the commit protocol (if supported by the underlying transaction system).
 - ◆ `false`—specifies that heuristic decisions should not be reported.
4. The `within_transaction()` call is needed at this point, because the `rollback_transaction()` function must only be called from within a transaction. If `rollback_transaction()` is called outside a transaction, it raises an exception.
5. If an exception is thrown, the transaction must be aborted by calling `IT_Bus::TransactionSystem::rollback_transaction()`.

http-conf Context Data Types

This appendix lists the http-conf context data types. You can use these C++ types in conjunction with the context API to set the properties of the HTTP transport plug-in programmatically.

C++ mapped classes

Example 251 shows the context data types that are generated when the `http-conf.xsd` schema is mapped to C++.

Example 251:*http-conf Context Data Types*

```
// C++
...
namespace IT_ContextAttributes
{
    ...
    class clientType
        : public IT_tExtensibilityElementData,
          public virtual IT_Bus::ComplexContentComplexType
    {
    public:
        ...
        clientType();
        clientType(const clientType & copy);
        virtual ~clientType();
        ...
        IT_Bus::Int * getSendTimeout();
    };
};
```

Example 251:*http-conf Context Data Types (Continued)*

```
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);

IT_Bus::Boolean *getAutoRedirect();
const IT_Bus::Boolean *getAutoRedirect() const;
void setAutoRedirect(const IT_Bus::Boolean * val);
void setAutoRedirect(const IT_Bus::Boolean & val);

IT_Bus::String *getUserName();
const IT_Bus::String *getUserName() const;
void setUserName(const IT_Bus::String * val);
void setUserName(const IT_Bus::String & val);

IT_Bus::String *getPassword();
const IT_Bus::String *getPassword() const;
void setPassword(const IT_Bus::String * val);
void setPassword(const IT_Bus::String & val);

IT_Bus::String *getAuthorizationType();
const IT_Bus::String *getAuthorizationType() const;
void setAuthorizationType(const IT_Bus::String * val);
void setAuthorizationType(const IT_Bus::String & val);

IT_Bus::String *getAuthorization();
const IT_Bus::String *getAuthorization() const;
void setAuthorization(const IT_Bus::String * val);
void setAuthorization(const IT_Bus::String & val);

IT_Bus::String *getAccept();
const IT_Bus::String *getAccept() const;
void setAccept(const IT_Bus::String * val);
void setAccept(const IT_Bus::String & val);

IT_Bus::String *getAcceptLanguage();
const IT_Bus::String *getAcceptLanguage() const;
void setAcceptLanguage(const IT_Bus::String * val);
void setAcceptLanguage(const IT_Bus::String & val);

IT_Bus::String *getAcceptEncoding();
```


Example 251: *http-conf* Context Data Types (Continued)

```
const IT_Bus::String *getAcceptEncoding() const;
void setAcceptEncoding(const IT_Bus::String * val);
void setAcceptEncoding(const IT_Bus::String & val);

IT_Bus::String *getContentType();
const IT_Bus::String *getContentType() const;
void setContentType(const IT_Bus::String * val);
void setContentType(const IT_Bus::String & val);

IT_Bus::String *getHost();
const IT_Bus::String *getHost() const;
void setHost(const IT_Bus::String * val);
void setHost(const IT_Bus::String & val);

Connection *getConnection();
const Connection *getConnection() const;
void setConnection(const Connection * val);
void setConnection(const Connection & val);

CacheControl *getCacheControl();
const CacheControl *getCacheControl() const;
void setCacheControl(const CacheControl * val);
void setCacheControl(const CacheControl & val);

IT_Bus::String *getCookie();
const IT_Bus::String *getCookie() const;
void setCookie(const IT_Bus::String * val);
void setCookie(const IT_Bus::String & val);

IT_Bus::String *getBrowserType();
const IT_Bus::String *getBrowserType() const;
void setBrowserType(const IT_Bus::String * val);
void setBrowserType(const IT_Bus::String & val);

IT_Bus::String *getReferer();
const IT_Bus::String *getReferer() const;
void setReferer(const IT_Bus::String * val);
void setReferer(const IT_Bus::String & val);

IT_Bus::String *getProxyServer();
const IT_Bus::String *getProxyServer() const;
void setProxyServer(const IT_Bus::String * val);
void setProxyServer(const IT_Bus::String & val);

IT_Bus::String *getProxyUserName();
```

Example 251:*http-conf Context Data Types (Continued)*

```

const IT_Bus::String *getProxyUserName() const;
void setProxyUserName(const IT_Bus::String * val);
void setProxyUserName(const IT_Bus::String & val);

IT_Bus::String *getProxyPassword();
const IT_Bus::String *getProxyPassword() const;
void setProxyPassword(const IT_Bus::String * val);
void setProxyPassword(const IT_Bus::String & val);

IT_Bus::String *getProxyAuthorizationType();
const IT_Bus::String *getProxyAuthorizationType() const;
void setProxyAuthorizationType(const IT_Bus::String *
val);
void setProxyAuthorizationType(const IT_Bus::String &
val);

IT_Bus::String *getProxyAuthorization();
const IT_Bus::String *getProxyAuthorization() const;
void setProxyAuthorization(const IT_Bus::String * val);
void setProxyAuthorization(const IT_Bus::String & val);

IT_Bus::Boolean *getUseSecureSockets();
const IT_Bus::Boolean *getUseSecureSockets() const;
void setUseSecureSockets(const IT_Bus::Boolean * val);
void setUseSecureSockets(const IT_Bus::Boolean & val);

IT_Bus::String *getClientCertificate();
const IT_Bus::String *getClientCertificate() const;
void setClientCertificate(const IT_Bus::String * val);
void setClientCertificate(const IT_Bus::String & val);

IT_Bus::String *getClientCertificateChain();
const IT_Bus::String *getClientCertificateChain() const;
void setClientCertificateChain(const IT_Bus::String *
val);
void setClientCertificateChain(const IT_Bus::String &
val);

IT_Bus::String *getClientPrivateKey();
const IT_Bus::String *getClientPrivateKey() const;
void setClientPrivateKey(const IT_Bus::String * val);
void setClientPrivateKey(const IT_Bus::String & val);

IT_Bus::String *getClientPrivateKeyPassword();

```

Example 251:*http-conf Context Data Types (Continued)*

```
        const IT_Bus::String *getClientPrivateKeyPassword()
const;
        void setClientPrivateKeyPassword(const IT_Bus::String *
val);
        void setClientPrivateKeyPassword(const IT_Bus::String &
val);

        IT_Bus::String *getTrustedRootCertificates();
const IT_Bus::String *getTrustedRootCertificates() const;
        void setTrustedRootCertificates(const IT_Bus::String *
val);
        void setTrustedRootCertificates(const IT_Bus::String &
val);
        ...
};
typedef IT_AutoPtr<clientType> clientTypePtr;

class Connection : public IT_Bus::AnySimpleType
{
public:
    ...
    Connection();
    Connection(const Connection & copy);
    Connection(const IT_Bus::String & value);
    virtual ~Connection();
    ...
    void set_value(const IT_Bus::String & value);
    const IT_Bus::String & get_value() const;
    ...
};
typedef IT_AutoPtr<Connection> ConnectionPtr;

class CacheControl : public IT_Bus::AnySimpleType
{
public:
    ...
    CacheControl();
    CacheControl(const CacheControl & copy);
    CacheControl(const IT_Bus::String & value);
    virtual ~CacheControl();
    ...
    void set_value(const IT_Bus::String & value);
    const IT_Bus::String & get_value() const;
    ...
};
```

Example 251:*http-conf Context Data Types (Continued)*

```

typedef IT_AutoPtr<CacheControl> CacheControlPtr;
...
class CacheControl_1 : public IT_Bus::AnySimpleType
{
public:
    ...
    CacheControl_1();
    CacheControl_1(const CacheControl_1 & copy);
    CacheControl_1(const IT_Bus::String & value);
    virtual ~CacheControl_1();

    ...
    void setvalue(const IT_Bus::String & value);
    const IT_Bus::String & getvalue() const;
    virtual IT_Reflect::Reflection* get_reflection()
        IT_THROW_DECL((IT_Reflect::ReflectException));
    ...
};

class serverType
: public IT_tExtensibilityElementData,
  public virtual IT_Bus::ComplexContentComplexType
{
public:
    ...
    serverType();
    serverType(const serverType & copy);
    virtual ~serverType();
    ...
    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);

    IT_Bus::Boolean *getSuppressClientSendErrors();
    const IT_Bus::Boolean *getSuppressClientSendErrors()
const;
    void setSuppressClientSendErrors(const IT_Bus::Boolean *
val);

```

Example 251:*http-conf Context Data Types (Continued)*

```
void setSuppressClientSendErrors(const IT_Bus::Boolean &
val);

IT_Bus::Boolean *getSuppressClientReceiveErrors();
const IT_Bus::Boolean *getSuppressClientReceiveErrors()
const;
void setSuppressClientReceiveErrors(const IT_Bus::Boolean
* val);
void setSuppressClientReceiveErrors(const IT_Bus::Boolean
& val);

IT_Bus::Boolean *getHonorKeepAlive();
const IT_Bus::Boolean *getHonorKeepAlive() const;
void setHonorKeepAlive(const IT_Bus::Boolean * val);
void setHonorKeepAlive(const IT_Bus::Boolean & val);

IT_Bus::Int *getMultiplexPoolSize();
const IT_Bus::Int *getMultiplexPoolSize() const;
void setMultiplexPoolSize(const IT_Bus::Int * val);
void setMultiplexPoolSize(const IT_Bus::Int & val);

IT_Bus::String *getRedirectURL();
const IT_Bus::String *getRedirectURL() const;
void setRedirectURL(const IT_Bus::String * val);
void setRedirectURL(const IT_Bus::String & val);

CacheControl_1 *getCacheControl();
const CacheControl_1 *getCacheControl() const;
void setCacheControl(const CacheControl_1 * val);
void setCacheControl(const CacheControl_1 & val);

IT_Bus::String *getContentLocation();
const IT_Bus::String *getContentLocation() const;
void setContentLocation(const IT_Bus::String * val);
void setContentLocation(const IT_Bus::String & val);

IT_Bus::String *getContentType();
const IT_Bus::String *getContentType() const;
void setContentType(const IT_Bus::String * val);
void setContentType(const IT_Bus::String & val);

IT_Bus::String *getContentEncoding();
const IT_Bus::String *getContentEncoding() const;
void setContentEncoding(const IT_Bus::String * val);
void setContentEncoding(const IT_Bus::String & val);
```

Example 251:*http-conf Context Data Types (Continued)*

```

IT_Bus::String *getServerType();
const IT_Bus::String *getServerType() const;
void setServerType(const IT_Bus::String * val);
void setServerType(const IT_Bus::String & val);

IT_Bus::Boolean *getUseSecureSockets();
const IT_Bus::Boolean *getUseSecureSockets() const;
void setUseSecureSockets(const IT_Bus::Boolean * val);
void setUseSecureSockets(const IT_Bus::Boolean & val);

IT_Bus::String *getServerCertificate();
const IT_Bus::String *getServerCertificate() const;
void setServerCertificate(const IT_Bus::String * val);
void setServerCertificate(const IT_Bus::String & val);

IT_Bus::String *getServerCertificateChain();
const IT_Bus::String *getServerCertificateChain() const;
void setServerCertificateChain(const IT_Bus::String *
val);
void setServerCertificateChain(const IT_Bus::String &
val);

IT_Bus::String *getServerPrivateKey();
const IT_Bus::String *getServerPrivateKey() const;
void setServerPrivateKey(const IT_Bus::String * val);
void setServerPrivateKey(const IT_Bus::String & val);

IT_Bus::String *getServerPrivateKeyPassword();
const IT_Bus::String *getServerPrivateKeyPassword()
const;
void setServerPrivateKeyPassword(const IT_Bus::String *
val);
void setServerPrivateKeyPassword(const IT_Bus::String &
val);

IT_Bus::String *getTrustedRootCertificates();
const IT_Bus::String *getTrustedRootCertificates() const;
void setTrustedRootCertificates(const IT_Bus::String *
val);
void setTrustedRootCertificates(const IT_Bus::String &
val);
...
};
typedef IT_AutoPtr<serverType> serverTypePtr;

```

Example 251:*http-conf Context Data Types (Continued)*

```
}
```


MQ-Series Context Data Types

This appendix lists the MQ-Series context data types. You can use these C++ types in conjunction with the context API to set the properties of the MQ transport plug-in programmatically.

C++ mapped classes

[Example 252](#) shows the context data types that are generated when the `mq.xsd` schema is mapped to C++.

Example 252:MQ-Series Context Data Types

```
// C++
...
namespace IT_ContextAttributes
{
    ...
    class transactionType : public IT_Bus::AnySimpleType
    {
    public:
        ...
        transactionType();
        transactionType(const transactionType & copy);
        transactionType(const IT_Bus::String & value);
        virtual ~transactionType();
        ...
        void setvalue(const IT_Bus::String & value);
        const IT_Bus::String & getvalue() const;
        ...
    };
};
```

Example 252:*MQ-Series Context Data Types (Continued)*

```

};
typedef IT_AutoPtr<transactionType> transactionTypePtr;

class correlationStyleType : public IT_Bus::AnySimpleType
{
public:
    ...
    correlationStyleType();
    correlationStyleType(const correlationStyleType & copy);
    correlationStyleType(const IT_Bus::String & value);
    virtual ~correlationStyleType();
    ...
    void setvalue(const IT_Bus::String & value);
    const IT_Bus::String & getvalue() const;
    ...
};
typedef IT_AutoPtr<correlationStyleType>
correlationStyleTypePtr;

class deliveryType : public IT_Bus::AnySimpleType
{
public:
    ...
    deliveryType();
    deliveryType(const deliveryType & copy);
    deliveryType(const IT_Bus::String & value);
    virtual ~deliveryType();
    ...
    void setvalue(const IT_Bus::String & value);
    const IT_Bus::String & getvalue() const;
    ...
};
typedef IT_AutoPtr<deliveryType> deliveryTypePtr;

class reportOptionType : public IT_Bus::AnySimpleType
{
public:
    ...
    reportOptionType();
    reportOptionType(const reportOptionType & copy);
    reportOptionType(const IT_Bus::String & value);
    virtual ~reportOptionType();
    ...
    void setvalue(const IT_Bus::String & value);
    const IT_Bus::String & getvalue() const;
};

```

Example 252:MQ-Series Context Data Types (Continued)

```
...
};
typedef IT_AutoPtr<reportOptionType> reportOptionTypePtr;

class formatType : public IT_Bus::AnySimpleType
{
public:
...
formatType();
formatType(const formatType & copy);
formatType(const IT_Bus::String & value);
virtual ~formatType();
...
void setvalue(const IT_Bus::String & value);
const IT_Bus::String & getvalue() const;
...
};
typedef IT_AutoPtr<formatType> formatTypePtr;
...
class usageStyleType : public IT_Bus::AnySimpleType
{
public:
...
usageStyleType();
usageStyleType(const usageStyleType & copy);
usageStyleType(const IT_Bus::String & value);
virtual ~usageStyleType();
...
void setvalue(const IT_Bus::String & value);
const IT_Bus::String & getvalue() const;
...
};
typedef IT_AutoPtr<usageStyleType> usageStyleTypePtr;

class accessModeType : public IT_Bus::AnySimpleType
{
public:
...
accessModeType();
accessModeType(const accessModeType & copy);
accessModeType(const IT_Bus::String & value);
virtual ~accessModeType();
...
void setvalue(const IT_Bus::String & value);
const IT_Bus::String & getvalue() const;
```

Example 252:*MQ-Series Context Data Types (Continued)*

```

    ...
};
typedef IT_AutoPtr<accessModeType> accessModeTypePtr;
...
class MQConnectionAttributesType
    : public IT_tExtensibilityElementData,
      public virtual IT_Bus::ComplexContentComplexType
{
public:
    ...
    MQConnectionAttributesType();
    MQConnectionAttributesType(const
MQConnectionAttributesType & copy);
    virtual ~MQConnectionAttributesType();
    ...
    IT_Bus::String * getQueueManager();
    const IT_Bus::String * getQueueManager() const;
    void setQueueManager(const IT_Bus::String * val);
    void setQueueManager(const IT_Bus::String & val);

    IT_Bus::String & getQueueName();
    const IT_Bus::String & getQueueName() const;
    void setQueueName(const IT_Bus::String & val);

    IT_Bus::String * getReplyQueueManager();
    const IT_Bus::String *getReplyQueueManager() const;
    void setReplyQueueManager(const IT_Bus::String * val);
    void setReplyQueueManager(const IT_Bus::String & val);

    IT_Bus::String *getReplyQueueName();
    const IT_Bus::String *getReplyQueueName() const;
    void setReplyQueueName(const IT_Bus::String * val);
    void setReplyQueueName(const IT_Bus::String & val);

    IT_Bus::String *getModelQueueName();
    const IT_Bus::String *getModelQueueName() const;
    void setModelQueueName(const IT_Bus::String * val);
    void setModelQueueName(const IT_Bus::String & val);

    IT_Bus::String *getAliasQueueName();
    const IT_Bus::String *getAliasQueueName() const;
    void setAliasQueueName(const IT_Bus::String * val);
    void setAliasQueueName(const IT_Bus::String & val);

    IT_Bus::String *getConnectionName();

```

Example 252:MQ-Series Context Data Types (Continued)

```
const IT_Bus::String *getConnectionName() const;
void setConnectionName(const IT_Bus::String * val);
void setConnectionName(const IT_Bus::String & val);

transactionType *getTransactional();
const transactionType *getTransactional() const;
void setTransactional(const transactionType * val);
void setTransactional(const transactionType & val);
...
};
typedef IT_AutoPtr<MQConnectionAttributesType>
MQConnectionAttributesTypePtr;

class mqClientType : public IT_tExtensibilityElementData ,
public virtual IT_Bus::ComplexContentComplexType
{
public:
...
mqClientType();
mqClientType(const mqClientType & copy);
virtual ~mqClientType();
...
IT_Bus::String *getQueueManager();
const IT_Bus::String *getQueueManager() const;
void setQueueManager(const IT_Bus::String * val);
void setQueueManager(const IT_Bus::String & val);

IT_Bus::String & getQueueName();
const IT_Bus::String & getQueueName() const;
void setQueueName(const IT_Bus::String & val);

IT_Bus::String *getReplyQueueManager();
const IT_Bus::String *getReplyQueueManager() const;
void setReplyQueueManager(const IT_Bus::String * val);
void setReplyQueueManager(const IT_Bus::String & val);

IT_Bus::String *getReplyQueueName();
const IT_Bus::String *getReplyQueueName() const;
void setReplyQueueName(const IT_Bus::String * val);
void setReplyQueueName(const IT_Bus::String & val);

IT_Bus::String *getModelQueueName();
const IT_Bus::String *getModelQueueName() const;
void setModelQueueName(const IT_Bus::String * val);
void setModelQueueName(const IT_Bus::String & val);
```

Example 252:*MQ-Series Context Data Types (Continued)*

```

usageStyleType *getUsageStyle();
const usageStyleType *getUsageStyle() const;
void setUsageStyle(const usageStyleType * val);
void setUsageStyle(const usageStyleType & val);

correlationStyleType *getCorrelationStyle();
const correlationStyleType *getCorrelationStyle() const;
void setCorrelationStyle(const correlationStyleType *
val);

void setCorrelationStyle(const correlationStyleType &
val);

accessModeType *getAccessMode();
const accessModeType *getAccessMode() const;
void setAccessMode(const accessModeType * val);
void setAccessMode(const accessModeType & val);

deliveryType *getDelivery();
const deliveryType *getDelivery() const;
void setDelivery(const deliveryType * val);
void setDelivery(const deliveryType & val);

transactionType *getTransactional();
const transactionType *getTransactional() const;
void setTransactional(const transactionType * val);
void setTransactional(const transactionType & val);

reportOptionType *getReportOption();
const reportOptionType *getReportOption() const;
void setReportOption(const reportOptionType * val);
void setReportOption(const reportOptionType & val);

formatType *getFormat();
const formatType *getFormat() const;
void setFormat(const formatType * val);
void setFormat(const formatType & val);

IT_Bus::String *getMessageID();
const IT_Bus::String *getMessageID() const;
void setMessageID(const IT_Bus::String * val);
void setMessageID(const IT_Bus::String & val);

IT_Bus::String *getCorrelationID();

```

Example 252:MQ-Series Context Data Types (Continued)

```
const IT_Bus::String *getCorrelationID() const;
void setCorrelationID(const IT_Bus::String * val);
void setCorrelationID(const IT_Bus::String & val);

IT_Bus::String *getApplicationData();
const IT_Bus::String *getApplicationData() const;
void setApplicationData(const IT_Bus::String * val);
void setApplicationData(const IT_Bus::String & val);

IT_Bus::String *getAccountingToken();
const IT_Bus::String *getAccountingToken() const;
void setAccountingToken(const IT_Bus::String * val);
void setAccountingToken(const IT_Bus::String & val);

IT_Bus::Boolean *getConvert();
const IT_Bus::Boolean *getConvert() const;
void setConvert(const IT_Bus::Boolean * val);
void setConvert(const IT_Bus::Boolean & val);

IT_Bus::String *getConnectionName();
const IT_Bus::String *getConnectionName() const;
void setConnectionName(const IT_Bus::String * val);
void setConnectionName(const IT_Bus::String & val);

IT_Bus::Boolean *getConnectionReusable();
const IT_Bus::Boolean *getConnectionReusable() const;
void setConnectionReusable(const IT_Bus::Boolean * val);
void setConnectionReusable(const IT_Bus::Boolean & val);

IT_Bus::Boolean *getConnectionFastPath();
const IT_Bus::Boolean *getConnectionFastPath() const;
void setConnectionFastPath(const IT_Bus::Boolean * val);
void setConnectionFastPath(const IT_Bus::Boolean & val);

IT_Bus::String *getAliasQueueName();
const IT_Bus::String *getAliasQueueName() const;
void setAliasQueueName(const IT_Bus::String * val);
void setAliasQueueName(const IT_Bus::String & val);

IT_Bus::String *getApplicationIdData();
const IT_Bus::String *getApplicationIdData() const;
void setApplicationIdData(const IT_Bus::String * val);
void setApplicationIdData(const IT_Bus::String & val);

IT_Bus::String *getApplicationOriginData();
```

Example 252:*MQ-Series Context Data Types (Continued)*

```

    const IT_Bus::String *getApplicationOriginData() const;
    void setApplicationOriginData(const IT_Bus::String *
val);
    void setApplicationOriginData(const IT_Bus::String &
val);

    IT_Bus::String *getUserIdentifier();
    const IT_Bus::String *getUserIdentifier() const;
    void setUserIdentifier(const IT_Bus::String * val);
    void setUserIdentifier(const IT_Bus::String & val);
    ...
};
typedef IT_AutoPtr<mqClientType> mqClientTypePtr;

class mqServerType
: public IT_tExtensibilityElementData,
  public virtual IT_Bus::ComplexContentComplexType
{
public:
    ...
    mqServerType();
    mqServerType(const mqServerType & copy);
    virtual ~mqServerType();
    ...
    IT_Bus::String *getQueueManager();
    const IT_Bus::String *getQueueManager() const;
    void setQueueManager(const IT_Bus::String * val);
    void setQueueManager(const IT_Bus::String & val);

    IT_Bus::String & getQueueName();
    const IT_Bus::String & getQueueName() const;
    void setQueueName(const IT_Bus::String & val);

    IT_Bus::String *getReplyQueueManager();
    const IT_Bus::String *getReplyQueueManager() const;
    void setReplyQueueManager(const IT_Bus::String * val);
    void setReplyQueueManager(const IT_Bus::String & val);

    IT_Bus::String *getReplyQueueName();
    const IT_Bus::String *getReplyQueueName() const;
    void setReplyQueueName(const IT_Bus::String * val);
    void setReplyQueueName(const IT_Bus::String & val);

    IT_Bus::String *getModelQueueName();
    const IT_Bus::String *getModelQueueName() const;

```


Example 252:MQ-Series Context Data Types (Continued)

```
void setModelQueueName(const IT_Bus::String * val);
void setModelQueueName(const IT_Bus::String & val);

usageStyleType *getUsageStyle();
const usageStyleType *getUsageStyle() const;
void setUsageStyle(const usageStyleType * val);
void setUsageStyle(const usageStyleType & val);

correlationStyleType *getCorrelationStyle();
const correlationStyleType *getCorrelationStyle() const;
void setCorrelationStyle(const correlationStyleType *
val);
void setCorrelationStyle(const correlationStyleType &
val);

accessModeType *getAccessMode();
const accessModeType *getAccessMode() const;
void setAccessMode(const accessModeType * val);
void setAccessMode(const accessModeType & val);

deliveryType *getDelivery();
const deliveryType *getDelivery() const;
void setDelivery(const deliveryType * val);
void setDelivery(const deliveryType & val);

transactionType *getTransactional();
const transactionType *getTransactional() const;
void setTransactional(const transactionType * val);
void setTransactional(const transactionType & val);

reportOptionType *getReportOption();
const reportOptionType *getReportOption() const;
void setReportOption(const reportOptionType * val);
void setReportOption(const reportOptionType & val);

formatType *getFormat();
const formatType *getFormat() const;
void setFormat(const formatType * val);
void setFormat(const formatType & val);

IT_Bus::String *getMessageID();
const IT_Bus::String *getMessageID() const;
void setMessageID(const IT_Bus::String * val);
void setMessageID(const IT_Bus::String & val);
```

Example 252:MQ-Series Context Data Types (Continued)

```
IT_Bus::String *getCorrelationID();
const IT_Bus::String *getCorrelationID() const;
void setCorrelationID(const IT_Bus::String * val);
void setCorrelationID(const IT_Bus::String & val);

IT_Bus::String *getApplicationData();
const IT_Bus::String *getApplicationData() const;
void setApplicationData(const IT_Bus::String * val);
void setApplicationData(const IT_Bus::String & val);

IT_Bus::String *getAccountingToken();
const IT_Bus::String *getAccountingToken() const;
void setAccountingToken(const IT_Bus::String * val);
void setAccountingToken(const IT_Bus::String & val);

IT_Bus::Boolean *getConvert();
const IT_Bus::Boolean *getConvert() const;
void setConvert(const IT_Bus::Boolean * val);
void setConvert(const IT_Bus::Boolean & val);

IT_Bus::String *getConnectionName();
const IT_Bus::String *getConnectionName() const;
void setConnectionName(const IT_Bus::String * val);
void setConnectionName(const IT_Bus::String & val);

IT_Bus::Boolean *getConnectionReusable();
const IT_Bus::Boolean *getConnectionReusable() const;
void setConnectionReusable(const IT_Bus::Boolean * val);
void setConnectionReusable(const IT_Bus::Boolean & val);

IT_Bus::Boolean *getConnectionFastPath();
const IT_Bus::Boolean *getConnectionFastPath() const;
void setConnectionFastPath(const IT_Bus::Boolean * val);
void setConnectionFastPath(const IT_Bus::Boolean & val);

IT_Bus::String *getApplicationIdData();
const IT_Bus::String *getApplicationIdData() const;
void setApplicationIdData(const IT_Bus::String * val);
void setApplicationIdData(const IT_Bus::String & val);

IT_Bus::String *getApplicationOriginData();
const IT_Bus::String *getApplicationOriginData() const;
void setApplicationOriginData(const IT_Bus::String *
val);
```

Example 252:MQ-Series Context Data Types (Continued)

```
        void setApplicationOriginData(const IT_Bus::String &
val);
        ...
};
typedef IT_AutoPtr<mqServerType> mqServerTypePtr;

class MQAttributesType
: public IT_tExtensibilityElementData,
  public virtual IT_Bus::ComplexContentComplexType
{
public:
    ...
    MQAttributesType();
    MQAttributesType(const MQAttributesType & copy);
    virtual ~MQAttributesType();
    ...
    IT_Bus::String *getQueueManager();
    const IT_Bus::String *getQueueManager() const;
    void setQueueManager(const IT_Bus::String * val);
    void setQueueManager(const IT_Bus::String & val);

    IT_Bus::String &getQueueName();
    const IT_Bus::String &getQueueName() const;
    void setQueueName(const IT_Bus::String & val);

    IT_Bus::String *getReplyQueueManager();
    const IT_Bus::String *getReplyQueueManager() const;
    void setReplyQueueManager(const IT_Bus::String * val);
    void setReplyQueueManager(const IT_Bus::String & val);

    IT_Bus::String *getReplyQueueName();
    const IT_Bus::String *getReplyQueueName() const;
    void setReplyQueueName(const IT_Bus::String * val);
    void setReplyQueueName(const IT_Bus::String & val);

    IT_Bus::String *getModelQueueName();
    const IT_Bus::String *getModelQueueName() const;
    void setModelQueueName(const IT_Bus::String * val);
    void setModelQueueName(const IT_Bus::String & val);

    usageStyleType *getUsageStyle();
    const usageStyleType *getUsageStyle() const;
    void setUsageStyle(const usageStyleType * val);
    void setUsageStyle(const usageStyleType & val);
```

Example 252:*MQ-Series Context Data Types (Continued)*

```

correlationStyleType *getCorrelationStyle();
const correlationStyleType *getCorrelationStyle() const;
void setCorrelationStyle(const correlationStyleType *
val);
void setCorrelationStyle(const correlationStyleType &
val);

accessModeType *getAccessMode();
const accessModeType *getAccessMode() const;
void setAccessMode(const accessModeType * val);
void setAccessMode(const accessModeType & val);

deliveryType *getDelivery();
const deliveryType *getDelivery() const;
void setDelivery(const deliveryType * val);
void setDelivery(const deliveryType & val);

transactionType *getTransactional();
const transactionType *getTransactional() const;
void setTransactional(const transactionType * val);
void setTransactional(const transactionType & val);

reportOptionType * getReportOption();
const reportOptionType * getReportOption() const;
void setReportOption(const reportOptionType * val);
void setReportOption(const reportOptionType & val);

formatType * getFormat();
const formatType * getFormat() const;
void setFormat(const formatType * val);
void setFormat(const formatType & val);

IT_Bus::Base64Binary * getMessageID();
const IT_Bus::Base64Binary * getMessageID() const;
void setMessageID(const IT_Bus::Base64Binary * val);
void setMessageID(const IT_Bus::Base64Binary & val);

IT_Bus::Base64Binary * getCorrelationID();
const IT_Bus::Base64Binary * getCorrelationID() const;
void setCorrelationID(const IT_Bus::Base64Binary * val);
void setCorrelationID(const IT_Bus::Base64Binary & val);

IT_Bus::String * getApplicationData();
const IT_Bus::String * getApplicationData() const;
void setApplicationData(const IT_Bus::String * val);

```

Example 252:MQ-Series Context Data Types (Continued)

```
void setApplicationData(const IT_Bus::String & val);

IT_Bus::String * getAccountingToken();
const IT_Bus::String * getAccountingToken() const;
void setAccountingToken(const IT_Bus::String * val);
void setAccountingToken(const IT_Bus::String & val);

IT_Bus::Boolean * getConvert();
const IT_Bus::Boolean * getConvert() const;
void setConvert(const IT_Bus::Boolean * val);
void setConvert(const IT_Bus::Boolean & val);

IT_Bus::String * getConnectionName();
const IT_Bus::String * getConnectionName() const;
void setConnectionName(const IT_Bus::String * val);
void setConnectionName(const IT_Bus::String & val);

IT_Bus::Boolean * getConnectionReusable();
const IT_Bus::Boolean * getConnectionReusable() const;
void setConnectionReusable(const IT_Bus::Boolean * val);
void setConnectionReusable(const IT_Bus::Boolean & val);

IT_Bus::Boolean * getConnectionFastPath();
const IT_Bus::Boolean * getConnectionFastPath() const;
void setConnectionFastPath(const IT_Bus::Boolean * val);
void setConnectionFastPath(const IT_Bus::Boolean & val);

IT_Bus::String * getAliasQueueName();
const IT_Bus::String * getAliasQueueName() const;
void setAliasQueueName(const IT_Bus::String * val);
void setAliasQueueName(const IT_Bus::String & val);

IT_Bus::String * getApplicationIdData();
const IT_Bus::String * getApplicationIdData() const;
void setApplicationIdData(const IT_Bus::String * val);
void setApplicationIdData(const IT_Bus::String & val);

IT_Bus::String * getApplicationOriginData();
const IT_Bus::String * getApplicationOriginData() const;
void setApplicationOriginData(const IT_Bus::String *
val);
void setApplicationOriginData(const IT_Bus::String &
val);

IT_Bus::String * getUserIdentifier();
```

Example 252:*MQ-Series Context Data Types (Continued)*

```

const IT_Bus::String * getUserIdentifier() const;
void setUserIdentifier(const IT_Bus::String * val);
void setUserIdentifier(const IT_Bus::String & val);

IT_Bus::Int * getBackoutCount();
const IT_Bus::Int * getBackoutCount() const;
void setBackoutCount(const IT_Bus::Int * val);
void setBackoutCount(const IT_Bus::Int & val);
...
};
typedef IT_AutoPtr<MQAttributesType> MQAttributesTypePtr;

class MQMessageAttributesType
: public IT_tExtensibilityElementData,
  public virtual IT_Bus::ComplexContentComplexType
{
public:
  ...
  MQMessageAttributesType();
  MQMessageAttributesType(const MQMessageAttributesType &
copy);
  virtual ~MQMessageAttributesType();
  ...
  correlationStyleType * getCorrelationStyle();
  const correlationStyleType * getCorrelationStyle() const;
  void setCorrelationStyle(const correlationStyleType *
val);
  void setCorrelationStyle(const correlationStyleType &
val);

  deliveryType * getDelivery();
  const deliveryType * getDelivery() const;
  void setDelivery(const deliveryType * val);
  void setDelivery(const deliveryType & val);

  reportOptionType * getReportOption();
  const reportOptionType * getReportOption() const;
  void setReportOption(const reportOptionType * val);
  void setReportOption(const reportOptionType & val);

  formatType * getFormat();
  const formatType * getFormat() const;
  void setFormat(const formatType * val);
  void setFormat(const formatType & val);

```

Example 252:MQ-Series Context Data Types (Continued)

```
IT_Bus::Base64Binary * getMessageID();
const IT_Bus::Base64Binary * getMessageID() const;
void setMessageID(const IT_Bus::Base64Binary * val);
void setMessageID(const IT_Bus::Base64Binary & val);

IT_Bus::Base64Binary * getCorrelationID();
const IT_Bus::Base64Binary * getCorrelationID() const;
void setCorrelationID(const IT_Bus::Base64Binary * val);
void setCorrelationID(const IT_Bus::Base64Binary & val);

IT_Bus::String * getApplicationData();
const IT_Bus::String * getApplicationData() const;
void setApplicationData(const IT_Bus::String * val);
void setApplicationData(const IT_Bus::String & val);

IT_Bus::String * getAccountingToken();
const IT_Bus::String * getAccountingToken() const;
void setAccountingToken(const IT_Bus::String * val);
void setAccountingToken(const IT_Bus::String & val);

IT_Bus::Boolean * getConvert();
const IT_Bus::Boolean * getConvert() const;
void setConvert(const IT_Bus::Boolean * val);
void setConvert(const IT_Bus::Boolean & val);

IT_Bus::String * getApplicationIdData();
const IT_Bus::String * getApplicationIdData() const;
void setApplicationIdData(const IT_Bus::String * val);
void setApplicationIdData(const IT_Bus::String & val);

IT_Bus::String * getApplicationOriginData();
const IT_Bus::String * getApplicationOriginData() const;
void setApplicationOriginData(const IT_Bus::String *
val);
void setApplicationOriginData(const IT_Bus::String &
val);

IT_Bus::String * getUserIdentifier();
const IT_Bus::String * getUserIdentifier() const;
void setUserIdentifier(const IT_Bus::String * val);
void setUserIdentifier(const IT_Bus::String & val);

IT_Bus::Int * getBackoutCount();
const IT_Bus::Int * getBackoutCount() const;
void setBackoutCount(const IT_Bus::Int * val);
```

Example 252: *MQ-Series Context Data Types (Continued)*

```
void setBackoutCount(const IT_Bus::Int & val);  
    ...  
};  
typedef IT_AutoPtr<MQMessageAttributesType>  
    MQMessageAttributesTypePtr;  
};
```


Index

Symbols

- ##any namespace constraint 341
- ##local namespace constraint 341
- ##other namespace constraint 342
- ##targetNamespace namespace constraint 341
- <bus-security security> 183
- <extension> tag 311
- <fault> tag 48
- <http-conf server> 182
- <http-conf:client> port extensor 214
- <http-conf:server> port extensor 217
- <i18n-context server> 182
- <mq client> 182
- <mq server> 182
- <restriction> tag 310
- <simpleContent> tag 310
- <soap header> element 190
- <soap:header> element 232

Numerics

- 16-bit characters 252

A

- abstract interface type 423
- _add_ref() function 445
- add_service() function 68, 69, 77
- All class 456
- all complex type
 - nillable example 375
- AllComplexType class 296
- all groups 296
- anonymous types
 - avoiding 306
- AnyHolder class 336
 - get_any_type() function 337
 - get_type() function 338
 - inserting and extracting atomic types 337

- inserting and extracting user types 337
 - set_any_type() function 337
- AnyType class 216, 219, 233, 337, 443
- AnyType type
 - printing 477
- anyType type 336
 - nillable 371
- anyURI type 334
- arithmetical operators
 - for integers 265
- arrays
 - multi-dimensional native 325
 - native 323
 - SOAP 400
- arrayType attribute 402
- array types
 - nillable elements 387
- artix.cfg file 89
- Artix Designer
 - and routing 135
- Artix foundation classes 24
- Artix locator
 - overview 149
- Artix namespaces 7
- Artix services
 - locator 153
- ART library 24
- assign() 413
- at() 413
- atomic types 247
 - nillable example 372
 - nillable types 371
- attach_thread() function
 - and suppressing propagation 559
- attributes
 - defining with anyURI 334
 - in extended types 315
 - mapping 299
 - optional 299
 - optional, C++ mapping 300
 - optional, example 300
 - prohibited 299
 - reflection of 495

- required 299
- required, C++ mapping 301
- required, example 301
- auto_ptr template 60

B

- backout count 569
- backout threshold 564, 567
 - configuring 569
- Base64Binary type 268
- base64Binary type
 - nillable 372
- begin_session() 173
- binary types 268
 - Base64Binary type 268
 - HexBinary type 268
- binding name
 - specifying to code generator 3
- bindings
 - configuration of 181
- boolean type
 - nillable 371
- BOQNAME attribute 569
- BOTHRESH attribute 569
- bounded sequences 431
- boxed value type 423
- building Artix applications 336
- BuiltInType class 451
- BuiltInType type 482
- Bus
 - add_service() function 68, 69, 77
- Bus library 24
- byte type
 - nillable 371

C

- C++ mapping
 - parameter order 37
 - parameters 36, 42
- callbacks
 - and routing 134
 - and threading 133
 - client implementation 141
 - ClientImpl servant class 143
 - client main function 141
 - demonstration 132
 - example scenario 133
 - overview 131
 - sample WSDL contract 138
 - server implementation 145
 - ServerImpl servant class 147
 - server main function 145
- casting
 - from plain pointer to Var 448
- checked facets 275
- Choice class 462
- choice complex type 306
- ChoiceComplexType class 292
- choice complex types 292
- Choice type 488
- clear() 413
- client
 - developing 14
 - proxy object 14
 - stub code, files 2
- client proxies
 - and multi-threading 82
 - and threading 81
- client stub code 2
- clone() function 87
- cloning
 - and transient servants 73
 - service for transient reference 125
- cloning services 72
- Code generation 2
- code generation
 - from the command line 3
 - impl flag 10
- code generator
 - command-line 3
 - files generated 2
- compare() 263, 266
- compilation
 - reflect flag 442
- compiler requirements 24
- compiling a context schema 224
- ComplexContent class 467
- complexContent tag 315
- ComplexContent type 493
- complex datatypes
 - generated files 2
- complex type
 - deallocating 59
 - deriving from simple 310
- ComplexType class 451
- complex types 288
 - assignment operators 57

- copying 57
 - deriving 313
 - nesting 306
 - recursive copying 58
- complexType tag 314, 315
- configuration
 - ORBname switch 159
- configuration contexts
 - example 209
 - header files 192
 - library 192
 - overview 181
 - pre-registered 192
 - registering 187
 - reply contexts 182
 - request contexts 182
 - schema-based 182
- ConnectException type 45
- const_cast_var casting operator 448
- ContextContainer class 215, 218, 232
- context containers
 - reply context 198
 - request context 198
- ContextCurrent class 198, 215, 218, 232
- ContextCurrent type 181
- context data
 - registering 232, 236
- context names 232
- context registry
 - registering 186
- ContextRegistry class 187, 188, 215, 218, 232
- ContextRegistry type 232
- contexts
 - client main function 214, 229
 - context name 232
 - ContextRegistry type 232
 - example 220
 - get_context() function 200
 - get_context_container() function 186
 - overview 180
 - overview of configuration contexts 181
 - overview of header contexts 184
 - protocols 184
 - register_context() function 186
 - registering a context type 186
 - reply_contexts() function 198
 - request_contexts() function 198
 - sample_schema 223
 - scenario description 222
 - schema, target namespace 224
 - server main function 217, 234
 - service implementation 237
 - set_context() function 200
 - stub files, generating 203
 - type factories for 186
 - user-defined data 203
- CORBA
 - abstract interface 423
 - any 424
 - basic types 424
 - boolean 424
 - boxed value 423
 - char 424
 - configuring internationalization 249
 - enum type 426
 - exception type 432
 - fixed 424
 - forward-declared interfaces 423
 - header context 185
 - local interface type 423
 - Object 424
 - registering a header context 190
 - sequence type 430
 - string 424
 - struct type 429
 - typedef 433
 - union type 427, 431
 - value type 423
 - wchar 424
 - wstring 424
- CORBA headers
 - and contexts 185
- D**
 - dateTime type
 - nillable 372
 - Date type 262
 - date type
 - nillable 372
 - decimal type
 - nillable 372
 - declaration specifiers 26
 - declspec option 26
 - Delivery attribute 565
 - derivation
 - by extension 310
 - by restriction 310
 - complex type from complex type 313

- get_derived() function 318
- get_simpleTypeValue() 312
- set_simpleTypeValue() 312
- DerivedSimpleType type 482
- DeserializationException type 45
- detach_thread() function
 - and suppressing propagation 559
- developing a server 10
- dispatch() function 86
- DLL
 - building stub libraries 26
- DLL library
 - building Artix stubs in a 5
- document/literal wrapped style
 - C++ default mapping 42
 - C++ mapping using -wrapped flag 43
 - declaring WSDL operations 40
 - overview 39
 - wrapped flag 6
- double type
 - nillable 371
- duration 287
- dynamic_cast_var casting operator 448

E

- ElementList class 470
- ElementList type 496
- elements
 - defining with anyURI 334
- embedded mode
 - compiling 24
 - linking 24
- encoding of SOAP array 406
- EndpointNotExist fault 156
- endpoint reference 104
- endpoints 151
 - registering with the locator 159
- end_session() 178
- ENTITIES 287
- ENTITIES type 302
- ENTITY 287
- ENTITY type 302
- enumeration facet 275
- enum type 426
- exception
 - raising a fault exception 48
- exception handling
 - CORBA mapping 432
- Exception type 45

- exception type 432
- extension
 - attributes defined in 315
 - deriving complex types 315
 - get_derived() function 318
 - holder types 318
- extension tag 315
- extensors
 - and configuration contexts 181

F

- facets 275
 - checked 275
- FaultException type 47
- fixed decimal
 - compare() 263
 - DigitIterator 264
 - is_negative() 263
 - left_most_digit() 263
 - number_of_digits() 263
 - past_right_most_digit() 263
 - round() 263
 - scale() 263
 - truncate() 263
- float type
 - nillable 371
- forward-declared interfaces 423
- fractionDigits facet 275

G

- GDay type 262
- gDay type
 - nillable 372
- generating code
 - complete sample application 19
- get_all_endpoints() 174
- get_any_namespace() function 348
- get_any_type() function 337
- get_attribute_value() function 495
- getBackoutCount() function 569
- get_base() function 486
- get_context() function 200, 233
- get_context_container() function 186
- get_current() function 215, 232, 238
- get_current_element() function 490
- get_derived() function 318
- get_discriminator() 428
- get_discriminator_as_uint() 428

get_element_name() function 489
 get_extents() 402, 407, 410
 get_item_name() 351
 get_max_occurs() 350
 get_max_occurs() function 357, 361
 get_min_occurs() 350
 get_min_occurs() function 357, 361
 get_namespace_constraints() function 348
 get_process_contents() function 348
 get_reference() function 127, 129
 get_reflected() function 443
 get_reflection() function 442
 get_simpleTypeValue() 312
 get_size() 351
 get_size() function 497
 get_type() function 338
 get_type_kind() function 443, 481, 489
 get_type_name() function 489
 get_value_kind() function 487
 GIOP

and Artix contexts 185
 service contexts 191
 GlobalBusORBPlugIn class 23
 GMonthDay type 262
 gMonthDay type
 nillable 372
 GMonth type 262
 gMonth type
 nillable 372
 GYearMonth type 262
 gYearMonth type
 nillable 372
 GYear type 262
 gYear type
 nillable 372

H

header contexts
 CORBA, registering 190
 example 220
 overview 184
 sample schema type 223
 SOAP, registering 189
 three-tier systems 240
 headers
 <soap:header> element 232
 HelloWorld port type 8
 HexBinary type 268
 hexBinary type

 nillable 372
 high water mark 89
 high_water_mark configuration variable 90
 holder types, and extension 318
 HTTP
 example port 15
 schema for transport 182
 http-conf:clientType type 211
 http-conf schema 210
 ReceiveTimeout 211
 SendTimeout 211

I

IANA character set 250
 IDL
 bounded sequences 431
 enum type 426
 exception type 432
 object references 436
 oneway operations 438
 sequence type 430
 struct type 429
 typedef 433
 union type 427, 431
 IDL attributes
 mapping to C++ 438
 IDL basic types 424
 IDL interfaces
 mapping to C++ 435
 IDL modules
 mapping to C++ 435
 IDL operations
 mapping to C++ 437
 parameter order 438
 return value 438
 IDL readonly attribute 439
 IDL-to-C++ mapping
 Artix and CORBA 422
 IDL types
 unsupported 423
 idl utility 422
 IDREF 287
 IDREFS 287
 IDREFS type 302
 imported schema
 C++ namespace for 4
 inheritance relationships
 between complex types 313
 init()

- ORBname parameter 164
- init() function 11, 14
- Initializing the Bus 11
- initial_threads configuration variable 90
- inout parameter ordering 38
- inout parameters 438
- in parameters 438
- input message 35, 40
- input parameters 35
- instance namespace 369
- integer
 - compare() 266
 - is_negative() 266
 - is_non_negative() 266
 - is_non_positive() 266
 - is_positive() 266
 - is_valid_integer() 266
 - to_string() 266
- Integer type 265
- integer type
 - nillable 372
- integer types
 - arithmetical operators 265
 - Integer type 265
 - maximum precision 265
 - NegativeInteger type 265
 - NonNegativeInteger type 265
 - NonPositiveInteger type 265
 - PositiveInteger type 265
- interceptors
 - configuration of 181
- International Components for Unicode 250
- internationalization
 - 16-bit characters 252
 - configuring 249
 - IANA character set 250
 - International Components for Unicode 250
 - narrow characters 251
 - plugins.codeset:char:ccs configuration variable 249
 - plugins.codeset:char:ncs configuration variable 249
 - plugins.codeset:wchar:ccs configuration variable 249
 - plugins.codeset:wchar:ncs configuration variable 249
 - plugins.soap:encoding configuration variable 249
 - schema 182
 - wchar_t characters 252
- interoperability
 - transaction propagation 556
- interposition
 - resource for 558
- int type
 - nillable 371
- InvalidRouteException type 46
- IOException type 45
- IONA foundation classes 24
- IOP
 - context ID 185
- IOP::ServiceId type 191
- IP ports
 - in cloned service 73
- is_empty() 410
- is_negative() 263, 266
- is_nil() function 374, 377, 384, 499
- is_non_negative() 266
- is_non_positive() 266
- is_positive() 266
- is_valid_integer() 266
- IT_AutoPtr template 60
- IT_Bus::AllComplexType 296
- IT_Bus::Any::get_any_namespace() function 348
- IT_Bus::Any::get_namespace_constraints() function 348
- IT_Bus::Any::get_process_contents() function 348
- IT_Bus::Any::set_any_data() function 344
- IT_Bus::Any::set_string_data() function 344
- IT_Bus::AnyList class 364
- IT_Bus::AnyType::get_reflection() function 442
- IT_Bus::AnyType::Kind type 443, 481
- IT_Bus::AnyType class 216, 219, 233, 443
- IT_Bus::AnyType type
 - printing 477
- IT_Bus::Base64Binary 268
- IT_Bus::Base64Binary type 268
- IT_Bus::BinaryBuffer 248
- IT_Bus::Boolean 247
- IT_Bus::Bus::register_servant() function 70
- IT_Bus::Bus::remove_service() function 71
- IT_Bus::Byte 247
- IT_Bus::ChoiceComplexType 292
- IT_Bus::ConnectException 45
- IT_Bus::ContextContainer::get_context() function 233
- IT_Bus::ContextContainer::request_contexts() function 233
- IT_Bus::ContextContainer class 215, 218, 232

IT_Bus::ContextCurrent::request_contexts()
 function 238
 IT_Bus::ContextCurrent class 198, 215, 218, 232
 IT_Bus::ContextRegistry::get_current()
 function 215, 232, 238
 IT_Bus::ContextRegistry::register_context()
 function 189, 190
 IT_Bus::ContextRegistry class 187, 188, 215, 218,
 232
 IT_Bus::ContextRegistry type 232
 IT_Bus::Date 248
 IT_Bus::DateTime 248, 261
 IT_Bus::Date type 262
 IT_Bus::Decimal 248, 263
 IT_Bus::Decimal::DigitIterator 264
 IT_Bus::DerivedSimpleType::get_base()
 function 486
 IT_Bus::DeserializationException 45
 IT_Bus::Double 247
 IT_Bus::Exception 45
 IT_Bus::Exception::message() 45
 IT_Bus::Exception type 45
 IT_Bus::FaultException 47
 IT_Bus::Float 247
 IT_Bus::GDay 248
 IT_Bus::GDay type 262
 IT_Bus::get_context_container() function 186
 IT_Bus::GlobalBusORBPlugin class 23
 IT_Bus::GMonth 248
 IT_Bus::GMonthDay 248
 IT_Bus::GMonthDay type 262
 IT_Bus::GMonth type 262
 IT_Bus::GYear 248
 IT_Bus::GYearMonth 248
 IT_Bus::GYearMonth type 262
 IT_Bus::GYear type 262
 IT_Bus::HexBinary 248, 268
 IT_Bus::HexBinary type 268
 IT_Bus::ID 248
 IT_Bus::init() 11, 14
 IT_Bus::Int 247
 IT_Bus::Integer 248
 IT_Bus::Integer type 265
 IT_Bus::IOException 45
 IT_Bus::Language 247
 IT_Bus::Long 247
 IT_Bus::Name 248
 IT_Bus::NCName 248
 IT_Bus::NegativeInteger 248
 IT_Bus::NegativeInteger type 265
 IT_Bus::NMToken 247
 IT_Bus::NMTokens 247
 IT_Bus::NonNegativeInteger 248
 IT_Bus::NonNegativeInteger type 265
 IT_Bus::NonPositiveInteger 248
 IT_Bus::NonPositiveInteger type 265
 IT_Bus::NormalizedString 247
 IT_Bus::PositiveInteger 248
 IT_Bus::PositiveInteger type 265
 IT_Bus::QName 248
 IT_Bus::QName type 259
 IT_Bus::RefCountedBase class 445
 IT_Bus::Reference class 105, 130
 IT_Bus::run() 12, 14
 IT_Bus::SequenceComplexType 289
 IT_Bus::SerializationException 45
 IT_Bus::Service::get_reference() function 127, 129
 IT_Bus::Service::register_servant() 68, 69, 77
 IT_Bus::Service::register_servant() function
 and transient servants 74
 IT_Bus::ServiceException 45
 IT_Bus::Short 247
 IT_Bus::shutdown() 16
 IT_Bus::SoapEncArrayT 402
 IT_Bus::String 247, 249
 IT_Bus::String::iterator 249
 IT_Bus::Time 248
 IT_Bus::Time type 262
 IT_Bus::Token 247
 IT_Bus::TransportException 45
 IT_Bus::UByte 247
 IT_Bus::UInt 247
 IT_Bus::ULong 247
 IT_Bus::UShort 247
 IT_Bus::Var template class 445
 IT_Bus namespace 7
 IT_Bus_Services::renewSessionFaultException 177
 iterators
 in IT_Vector 414
 IT_FixedPoint class 263
 IT_Reflect::All class 456
 IT_Reflect::BuiltInType::get_value_kind()
 function 487
 IT_Reflect::BuiltInType::ValueKind type 487
 IT_Reflect::BuiltInType class 451
 IT_Reflect::BuiltInType type 482
 IT_Reflect::Choice::get_current_element()
 function 490

IT_Reflect::Choice class 462
 IT_Reflect::Choice type 488
 IT_Reflect::ComplexContent class 467
 IT_Reflect::ComplexContent type 493
 IT_Reflect::ComplexType class 451
 IT_Reflect::DerivedSimpleType type 482
 IT_Reflect::ElementList::get_size() function 497
 IT_Reflect::ElementList class 470
 IT_Reflect::ElementList type 496
 IT_Reflect::ModelGroup class 451
 IT_Reflect::ModelGroup type 488
 IT_Reflect::Nillable::is_nil() function 499
 IT_Reflect::Nillable class 473
 IT_Reflect::Nillable type 498
 IT_Reflect::Reflection::get_reflected() function 443
 IT_Reflect::Reflection::get_type_kind() function 481, 489
 IT_Reflect::Reflection::get_type_name() function 489
 IT_Reflect::Reflection class 442, 451
 IT_Reflect::Sequence class 459
 IT_Reflect::SimpleContent class 465
 IT_Reflect::SimpleContent type 491
 IT_Reflect::SimpleType class 451
 IT_Reflect::ValueRef template type 443
 IT_Reflect::Value template class 452
 IT_Routing::InvalidRouteException 46
 IT_UString class 249
 IT_Vectorof class
 resize() 413
 IT_Vector class
 assign() 413
 at() 413
 clear() 413
 converting to 327
 differences from std::vector 413
 iterators 414
 operations 416
 overview 412
 resize() 413
 IT_Vector template class
 and AnyList type 364
 IT_WSDL namespace 7

K

Kind type 481

L

lax 342
 leaks
 avoiding 60
 left_most_digit() 263
 length() 253
 length facet 275
 libraries
 Artix foundation classes 24
 ART library 24
 Bus 24
 IONA foundation classes 24
 license
 display current 5
 linker requirements 24
 load balancing
 with the locator 150
 local interface type 423
 locator
 binding and protocol 153
 demonstration code 151
 embedded deployment 151
 EndpointNotExist fault 156
 load balancing 150, 152
 LocatorService port type, C++ mapping 156
 lookupEndpointResponse type 156
 lookupEndpoint type 155
 reading a reference from 161
 registering endpoints 159
 standalone deployment 151
 WSDL contract 153
 locator, Artix 149
 locator_endpoint plug-in 159
 LocatorService port type 156
 logical contract
 and servants 65
 long type
 nillable 371
 lookupEndpointResponse type 156
 lookupEndpoint type 155
 low water mark 89
 low_water_mark configuration variable 90

M

makefile
 generating with wsdltocpp 4
 mapping
 IDL attributes 438

- IDL interfaces 435
- IDL modules 435
- IDL operations 437
- IDL to C++ 422
- maxExclusive facet 275
- maxInclusive facet 275
- maxLength facet 275
- maxOccurs 323, 350
- max_size() 413
- memory management 51
 - client side 53
 - copying and assignment 57
 - deallocating 59
 - reflection 445
 - rules 52
 - server side 54
 - smart pointers 60
- message() function 45
- message headers
 - and contexts 184
- messages
 - input 35, 40
 - output 35, 41
- minExclusive facet 275
- minInclusive facet 275
- minLength facet 275
- minOccurs 350
- ModelGroup class 451
- ModelGroup type 488
- MQ-Series
 - BOQNAME attribute 569
 - BOTHRESH attribute 569
 - runmqsc command-line tool 569
 - schema for transport 182
- MQ transactions 562
 - backout count 569
 - backout threshold 564, 567, 569
 - Delivery attribute 565
 - synchronous invocation 566
 - Transactional attribute 565
- multi-dimensional native arrays 325
- multiple occurrences
 - printing with reflection 496
- multi-threaded threading model 83
- multi-threading
 - client side 81
 - server side 83

N

- namespace
 - for generated C++ code 3
- namespace constraints
 - accessing 347
 - xsd:any element 341
- namespace prefix 259
- namespaces
 - IT_Bus 7
 - IT_WSDL 7
 - using in C++ 7
- namespace URI
 - and QName type 259
 - anyURI type 334
 - exclude from code generation 4
 - include in code generation 4
- narrow characters 251
- native arrays 323
- NegativeInteger type 265
- negativeInteger type
 - nillable 372
- nesting complex types 306
- nillable atomic member elements 378
- Nillable class
 - and reflection 473
- NillablePtr template class 384
- Nillable type 498
- nillable type
 - reflection 473
- nillable types 378
 - atomic type, example 372
 - atomic types 371
 - IT_Bus::NillableValue 369
 - nillable array elements 387
 - NillablePtr template class 384
 - nillable user-defined member elements 382
 - overview 368
 - syntax 369
 - user-defined types 375
 - xsi:nil attribute 369
- NillableValue class 369
- nmake
 - generating makefile for 4
- NMTOKENS type 302
- NMTOKEN type 302
- NonNegativeInteger type 265
- nonNegativeInteger type
 - nillable 372
- NonPositiveInteger type 265

nonPositiveInteger type
 nillable 372
 NOTATION 287
 NOTATION type 302
 number_of_digits() 263

O

object references
 mapping to C++ 436
 occurrence constraints 357, 361
 and reflection 470
 AnyList class 364
 get_item_name() 351
 get_max_occurs() 350
 get_max_occurs() function 357, 361
 get_min_occurs() 350
 get_size() 351
 in all groups 296
 in choice groups 292
 in sequence groups 289
 overview of 350
 sequence 355, 359
 set_size() 350
 set_size() function 357, 361
 xsd:any element 341
 xsd:any type 363
 offset attribute 411
 oneway invocations
 and MQ transactions 563
 oneway operations
 in IDL 438
 operations
 declaring 35, 40
 optional attributes 299
 -ORBname, parameter to IT_Bus::init() 164
 -ORBname command-line parameter 159
 -ORBname command-line switch 89
 orb_plugins list 106
 order of parameters 37
 OTS Lite
 limitations on using 558
 out parameters 438
 output directory
 specifying to code generator 3
 output message 35, 41
 output parameters 35

P

parameters
 in IDL-to-C++ mapping 438
 parsing
 WSDL model 107
 partially transmitted arrays 411
 past_right_most_digit() 263
 pattern facet 275
 PerInvocation threading model 85
 threading
 PerInvocation threading model
 87
 per-port threading model 84, 86
 PerThread threading model 85, 87
 physical contract
 and servants 65
 plug-in
 servant registration 21
 servant registration code 5
 plug-ins
 locator_endpoint 159
 plugins:codeset:char:ccs configuration variable 249
 plugins:codeset:char:ncs configuration variable 249
 plugins:codeset:wchar:ccs configuration
 variable 249
 plugins:codeset:wchar:ncs configuration
 variable 249
 plugins:sm_simple_policy:max_session_timeout 17
 3
 plugins:sm_simple_policy:min_session_timeout 173
 plugins:soap:encoding configuration variable 249
 port
 specifying on the client side 14
 port extensors
 <bus-security
 security> 183
 <http-conf
 server> 182
 <http-conf:client> 214
 <http-conf:server> 217
 <i18n-context
 server> 182
 <mq
 client> 182
 server> 182
 and configuration contexts 181
 ports
 activating all together 67

- activating individually 68, 69, 77
- activating with `register_servant()` 67
- and endpoints 151
- port type
 - specifying to code generator 3
- PositiveInteger type 265
- positivelnteger type
 - nillable 372
- `print_atom` template function 486
- Printer class 477
- printing Choice type 488
- printing DerivedSimpleType type 482
- `print_random` demonstration 476
- `print_value()` template function 486
- processContents attribute 342
 - `get_process_contents()` function 348
 - lax 342
 - skip 342
 - strict 342
- prohibited attributes 299
- protocols
 - and contexts 184
- proxification 134
 - definition 136
- proxy
 - initializing from reference 130
- proxy object
 - and multi-threading 82
 - constructors 14
- proxy objects
 - constructor with reference argument 16

Q

- QName type 259
 - equality testing 260
 - nillable 371

R

- recursive copying 58
- recursive deallocating 59
- recursive descent parsing 442
- RefCountedBase class 445
- reference
 - C++ representation 105
 - contents 105
 - to an endpoint 104
 - XML schema for 105
- Reference class 105

- reference counting 445
 - `_add_ref()` function 445
 - `_remove_ref()` function 445
 - Var assignment 446
- references
 - and WSDL publish plug-in 108
 - callbacks, overview 131
 - cloning from a service 125
 - CORBA mapping 436
 - creating 126
 - `get_reference()` function 129
 - importing the XML schema 123
 - IT_Bus::Reference class 130
 - looking up in the locator 151
 - programming with 116
 - proxy constructor 16, 130
 - reading from the locator 161
 - `register_transient_servant()` function 129
 - XML schema 105, 117
 - XML type 117
- references:Reference type 123
- reflect flag 6, 442
- reflection
 - All class 456
 - API overview 450
 - attributes 495
 - casting 448
 - Choice class 462
 - ComplexContent class 467
 - converting a built-in type 443
 - converting reflection to AnyType 443
 - ElementList class 470
 - example 476
 - `get_attribute_value()` function 495
 - `get_base()` function 486
 - `get_current_element()` function 490
 - `get_element_name()` function 489
 - `get_size()` function 497
 - `get_type_kind()` function 443, 481, 489
 - `get_type_name()` function 489
 - `get_value_kind()` function 487
 - `is_nil()` function 499
 - Kind type 443, 481
 - memory management 445
 - multiple occurrences 496
 - Nillable class 473
 - occurrence constraints 470
 - overview 442
 - `print_atom` template function 486

- Printer class 477
- printing BuiltInType type 482
- printing ComplexContent type 493
- printing ElementList type 496
- printing ModelGroup type 488
- printing Nillable type 498
- printing SimpleContent type 491
- print_value() template function 486
- RefCountedBase class 445
 - reflect flag 6, 442
- Sequence class 459
- SimpleContent class 465
- simple types 452
 - type descriptions 443
- ValueKind type 487
- Value template class 452
- Var template class 445
- Reflection class 442, 451
- register_context() function 184, 186, 187, 188, 189, 190, 232, 236
- register_servant() function 67, 70, 127
 - and transient servants 74
- register_transient_servant() function 75, 79, 129
- reliable messaging
 - and transactions 562
- _remove_ref() function 445
- remove_service() function 71
- renew_session() 177
- reply context container 198
- reply contexts
 - and configuration contexts 182
- reply_contexts() function 198
- reply message
 - document/literal wrapped 40
- request context
 - propagating automatically 241
- request context container 198
- request contexts
 - and configuration contexts 182
- request_contexts() function 182, 198, 233, 238
- request message
 - document/literal wrapped 39
- required attributes 299
- resize() 413
- restriction tag 314
- round() 263
- router contract 135
- routing
 - and callbacks 134

- Artix Designer 135
 - proxification 136
- run() function 12, 14
- runmqsc command-line tool 569
- Running the Bus 12

S

- sample context schema 223
- scale() 263
- schemas
 - and configuration contexts 182
 - context, example 223
 - for references 105
 - http-conf schema 210
 - HTTP transport 182
 - internationalization 182
 - MQ-series transport 182
 - pre-registered contexts, for 192
- Sequence class 459
- sequence complex type 306
- SequenceComplexType class 289
- sequence complex types 289
 - and arrays 323
- sequence type 430
 - get_max_occurs() function 357, 361
 - get_min_occurs() function 357, 361
 - occurrence constraints 355, 359
 - set_size() function 357, 361
- Serialization type 45
- Serialized threading model 87
- serialized threading model 84
- servant
 - and threading models 85
 - registration in plug-in 5
 - static, example 70
- servants
 - add_service() function 68, 69, 77
 - clone() function 87
 - dispatch() function 86
 - registering 64
 - register_servant() function 67
 - static, registering 65
 - transient, registering 72
 - wrapper, registering 87
 - wrapper classes 86
- server
 - developing 10
 - implementation class 10
 - main() function 11

- skeleton code, files 2
 - server skeleton code 2
 - service
 - specifying on the client side 14
 - Service::register_servant() 68, 69, 77
 - service contexts
 - and CORBA 185
 - context ID 191
 - IOP context ID 185
 - ServiceException type 45
 - service name
 - specifying to code generator 3
 - services
 - cloning 72
 - cloning, IP ports 73
 - SessionManagerClient 172
 - set_any_data() function 344
 - set_any_type() function 337
 - set_context() function 200
 - set_simpleTypeValue() 312
 - set_size() 350
 - set_size() function 357, 361
 - set_string_data() function 344
 - short type
 - nillable 371
 - shutdown() function 16
 - Shutting the Bus down 13
 - SimpleContent class 465
 - SimpleContent type 491
 - SimpleType class 451
 - simple types
 - deriving by restriction 275
 - skeleton code
 - files 2
 - generating with wsdltocpp 4
 - skip 342
 - smart pointers 60
 - Var type 489
 - SOAP
 - header context 184
 - internationalization 249
 - registering a header context 189
 - SOAP arrays 400
 - encoding 406
 - get_extents() 402, 407
 - multi-dimensional 405
 - one-dimensional 402
 - partially transmitted 411
 - sparse 408
 - syntax 401
 - SOAP bindings 153
 - SOAP-ENC:Array type 401
 - SOAP-ENC:offset attribute 411
 - SoapEncArrayT class 402
 - SOAPHeaderInfo type 223
 - SOAP headers
 - and contexts 184
 - sparse arrays 408
 - get_extents() 410
 - initializing 409
 - is_empty() 410
 - static_cast_var casting operator 448
 - static servant
 - definition 65
 - static servants 65
 - register_servant() function 127
 - std::vector class 412
 - strict 342
 - strings
 - iterator 249
 - IT_UString class 249
 - length() 253
 - String type
 - conversion functions 252
 - string type
 - nillable 371
 - Stroustrup, Bjarne 253
 - struct type 429
 - stub code
 - files 2
 - stub libraries
 - building on Windows 26
 - stubs
 - DLL library, packaging as 5
 - synchronous invocation
 - and MQ transactions 566
- ## T
- target namespace
 - for a context schema 224
 - threading
 - and callbacks 133
 - and configuration contexts 181
 - and ContextCurrent type 181
 - client proxy in two threads 81
 - multi-threaded model 83
 - overview 80
 - PerlInvocation threading model 85

- per-port threading model 84, 86
- PerThread threading model 85, 87
- Serialized threading model 87
- serialized threading model 84
- work queue 85
- threading model
 - default 83
 - default, for servants 77
 - default for servant 69
- thread pool
 - configuration settings 89
 - initial threads 89
- thread_pool:high_water_mark configuration variable 90
- thread_pool:initial_threads configuration variable 90
- thread_pool:low_water_mark configuration variable 90
- time
 - Date type 262
 - GDay type 262
 - GMonthDay type 262
 - GMonth type 262
 - GYearMonth type 262
 - GYear type 262
 - Time type 262
- time type
 - nillable 372
- to_string() 266
- totalDigits facet 275
- Transactional attribute 565
- transaction contexts 556
- transaction propagation 556
 - suppressing, how to 559
- transactions
 - compatibility with CORBA OTS 521
- transient servants 72
 - registering 74
- TransportException type 45
- transports
 - configuration of 181
- truncate() 263
- Tuxedo
 - example port 15
- typedef 433
- type factories
 - and contexts 186

U

- union type 427, 431

- unsignedByte type
 - nillable 371
- unsignedInt type
 - nillable 371
- unsignedLong type
 - nillable 371
- unsignedShort type
 - nillable 371
- unsupported IDL types 423
- UsageStyle attribute 568
- user defined exceptions
 - propagation 47
- user-defined types
 - nillable 375

V

- ValueKind type 487
- ValueRef template type 443
- Value template class 452
- value type 423
- Var template class 445
- Var type
 - assignment 446
 - casting, from plain pointer to Var 448
 - casting, from Var to Var 448
 - const_cast_var casting operator 448
 - dynamic_cast_var casting operator 448
 - static_cast_var casting operator 448

W

- wchar_t characters 252
- wchar type 423
- whiteSpace facet 275
- wildcarding types 333
 - anyURI type 334
 - xsd:any element 341
- work queue 85
 - wrapped flag 6, 43
- wrapped parameters
 - wrapped flag 6
- wrapper servants 86, 87
- WSDL
 - anyType syntax 336
 - atomic types 247
 - attributes 299
 - binary types 268
 - complex types 288
 - deriving by restriction 275

- wSDL:arrayType attribute 402
- WSDL contract
 - location of 15
- WSDL facets 275
- WSDL faults 432
- WSDL model 107
 - and multiple Bus instances 112
- WSDL publish plug-in 106
 - WSDL model 107
- wSDL_publish plug-in 106
- wsdltocpp
 - command-line options 3
 - command-line switches 3
 - files generated 2
 - XML schemas, generating from 203
- wsdltocpp compiler 224
 - generating an application 19
- wsdltocpp utility 336, 422
 - declspec option 26
 - reflect flag 442
 - wrapped flag 43
- wstring type 423

X

- XML schema
 - wildcarding types 333
- xsd:any element 341
 - namespace constraint 341
 - occurrence constraints 341
 - process contents attribute 342
- xsd:any type
 - AnyList class 364
 - occurrence constraints 363
- xsd:anyURI type 334
- xsd:boolean 276
- xsd:dateTime type 261
- xsd:day schema type 262
- xsd:decimal type 263
- xsd:duration 287
- xsd:ENTITIES 287, 302
- xsd:ENTITY 287, 302
- xsd:IDREF 287
- xsd:IDREFS 287, 302
- xsd:NMTOKEN 302
- xsd:NMTOKENS 302
- xsd:NOTATION 287, 302
- xsd:time schema type 262
- xsi:nil attribute 369
- xsi namespace 369

