



Developing Artix Applications in C++

Version 1.3, December 2003

IONA, IONA Technologies, the IONA logo, Orbix, Orbix/E, ORBacus, Artix, Orchestrator, Mobile Orchestrator, Enterprise Integrator, Adaptive Runtime Technology, Transparent Enterprise Deployment, and Total Business Integration 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 Technologies PLC 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 book. This publication and features described herein are subject to change without notice.

Copyright © 2003 IONA Technologies PLC. All rights reserved.

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

Updated: 19-Dec-2003

M 3 1 8 2

Contents

List of Tables	vii
Preface	ix
Chapter 1 Developing Artix Enabled Clients and Servers	1
Generating Stub and Skeleton Code	2
C++ Namespaces	5
Defining a WSDL Interface	6
Developing a Server	8
Developing a Client	12
Compiling and Linking an Artix Application	17
Building Artix Stub Libraries on Windows	19
Chapter 2 Artix Programming Considerations	21
Operations and Parameters	22
Exceptions	26
Non-Propagating Exceptions	27
Propagating Exceptions	29
Memory Management	33
Managing Parameters	34
Assignment and Copying	39
Deallocating	41
Smart Pointers	42
Implementing a Server Factory	46
Multi-Threading	51
Client Threading Issues	52
Server Threading Models	54
Changing the Server Threading Model	58
Chapter 3 Artix References	61
Introduction to References	62
The <code>IT_Bus::Reference</code> Class	65
Using the Artix Locator	66

Overview of the Locator	67
Locator WSDL	69
Registering Endpoints with the Locator	75
Reading a Reference from the Locator	76
Pausing and Resuming Endpoints	80
Chapter 4 Using Sessions in Artix	83
Introduction to Session Management in Artix	84
Registering a Server with the Session Manager	87
Working with Sessions	90
Chapter 5 Transactions in Artix	99
Introduction to Transactions	100
Transaction API	102
Client Example	104
Chapter 6 Message Attributes	107
Introduction to Message Attributes	108
Schemas	111
Name-Value API	113
Transport-Specific API	117
Using Message Attributes in a Client	120
Using Message Attributes in a Server	123
Chapter 7 Dynamic Configuration	127
Introduction to Dynamic Configuration	128
Dynamically Allocating IP Ports	130
Chapter 8 Artix Data Types	135
Simple Types	136
Atomic Types	137
String Type	138
QName Type	139
Date and Time Types	141
Decimal Type	142
Binary Types	144
Deriving Simple Types by Restriction	146

Unsupported Simple Types	149
Complex Types	150
Sequence Complex Types	151
Choice Complex Types	154
All Complex Types	158
Attributes	161
Nesting Complex Types	163
Deriving a Complex Type from a Simple Type	167
Occurrence Constraints	170
Arrays	174
anyType Type	179
Nillable Types	184
Introduction to Nillable Types	185
Nillable Atomic Types	187
Nillable User-Defined Types	191
Nested Atomic Type Nillable Elements	194
Nested User-Defined Nillable Elements	198
Nillable Elements of an Array	202
SOAP Arrays	205
Introduction to SOAP Arrays	206
Multi-Dimensional Arrays	210
Sparse Arrays	213
Partially Transmitted Arrays	216
IT_Vector Template Class	217
Introduction to IT_Vector	218
Summary of IT_Vector Operations	221
Chapter 9 Artix IDL to C++ Mapping	225
Introduction to IDL Mapping	226
IDL Basic Type Mapping	228
IDL Complex Type Mapping	229
IDL Module and Interface Mapping	238
Index	243

CONTENTS

List of Tables

Table 1: Artix Import Libraries for Linking with an Application	17
Table 2: Artix Exception Error Codes	27
Table 3: Pattern of create_server() Calls in Various Threading Models	59
Table 4: Transport Schemas with Message Attributes	111
Table 5: Simple Schema Type to Simple Bus Type Mapping	137
Table 6: Member Fields of IT_Bus::DateTime	141
Table 7: Operators Supported by IT_Bus::Decimal	142
Table 8: Schema to Bus Mapping for the Binary Types	144
Table 9: Nillable Atomic Types	187
Table 10: Member Functions Not Defined in IT_Vector	218
Table 11: Member Types Defined in IT_Vector<T>	221
Table 12: Iterator Member Functions of IT_Vector<T>	222
Table 13: Element Access Operations for IT_Vector<T>	222
Table 14: Stack Operations for IT_Vector<T>	222
Table 15: List Operations for IT_Vector<T>	223
Table 16: Other Operations for IT_Vector<T>	223
Table 17: Artix Mapping of IDL Basic Types to C++	228

LIST OF TABLES

Preface

Audience

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.

Related documentation

The document set for Artix includes the following:

- *Getting Started with Artix*
- *Deploying and Managing Artix Solutions*
- *Designing Artix Solutions*
- *Artix Tutorial Guide*

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

Reading path

If you are new to Artix, you should read the documentation in the following order:

1. *Getting Started with Artix*
The getting started book describes the basic concepts behind Artix. It also provides details on installing the system and a detailed walk through for developing a C++ client for a Web Service.
2. *Deploying and Managing Artix Solutions*
This guide describes deploying Artix enabled systems. It provides detailed examples for a number of typical use cases.
3. *Designing Artix Solutions*

This guide describes the process of describing your systems and integrations in an Artix contract.

4. *Artix Tutorial Guide*

The tutorial guides you through programming Artix applications against all of the supported transports.

5. GUI Online Help

The Artix design tools have context sensitive on-line help the provides information specific to the tools that you are using.

Help resources

If you need help with this or any other IONA products, contact IONA at support@iona.com. Comments on IONA documentation can be sent to docs-support@iona.com.

Additional resources

The IONA knowledge base contains helpful articles, written by IONA experts, about the Orbix and other products. You can access the knowledge base at the following location:

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

The IONA update center contains the latest releases and patches for IONA products:

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

Typographical conventions

This guide uses the following typographical conventions:

Constant width Constant 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 `CORBA::Object` class.

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

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

Italic

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

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/your_name
```

Note: Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

Keying conventions

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
%	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 DOS, Windows NT, Windows 95, or Windows 98 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.
	A vertical bar separates items in a list of choices enclosed in {} (braces) in format and syntax descriptions.

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 5
Developing a Server	page 8
Developing a Client	page 12
Compiling and Linking an Artix Application	page 17
Building Artix Stub Libraries on Windows	page 19

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 is similar to code generated by a CORBA IDL compiler. There are two major differences between CORBA generated code and Artix generated code:

- Artix generated code is not restricted to using IIOP and therefore contains generic code that is compatible with a multitude of transports.
- Artix maps WSDL types to C++ using a proprietary WSDL-to-C++ mapping. The resulting types are very different from those generated by an IDL-to-C++ compiler.

Generated files

The Artix code generator produces seven files from the Artix contract. They are named according to the port type name specified in the logical portion of the Artix contract. The files are as follows:

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.

PortTypeNameTypes.h and *PortTypeNameTypes.cxx* define the complex datatypes defined in the contract.

PortTypeNameService.h and *PortTypeNameService.cxx* are the server-side skeleton code to implement the service defined in the contract.

PortTypeNameClient.h and *PortTypeNameClient.cxx* are the client-side stubs for implementing a client to use the service defined by the contract.

If the contract specifies more than one port type, code will be generated for each port type defined.

Generating code from the command line

You can generate code at the command line using the command:

```
wsdltocpp WSDL_URL [-i port_type] [-e web_service_name] [-t port]
[-b binding_name] [-d output_dir] [-n namespace] [--impl [-m
{NMAKE | UNIX}]] [-f] [-sample] [-v] [--license] [--declspec
declspec] [--all] [--?] [--flags] [--nimport=namespace]
```

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.
<code>-e web_service_name</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.
<code>-t port</code>	Specifies the name of the port for which code is generated. The default is to use the first port listed in the contract.
<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 namespace</code>	Specifies the C++ namespace to use for the generated code.
<code>-impl</code>	Generates the skeleton code for implementing the server defined by the contract.
<code>-m {NMAKE UNIX}</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). For example, the options, <code>-impl -m NMAKE</code> , would generate a Windows makefile.
<code>-f</code>	<i>Deprecated</i> —No longer used (was needed to support routing in earlier versions).
<code>-sample</code>	Generates code for a sample implementation of a client and a server.

<code>-v</code>	Displays the version of the tool.
<code>-license</code>	Displays the currently available licenses.
<code>-declspec <i>declspec</i></code>	Creates NT 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 19 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.
<code>-?</code>	Displays help on using the command line tool.
<code>-flags</code>	Displays detailed information about the options.
<code>-nimport=<i>namespace</i></code>	Specifies the namespace under which code from imported schema is generated. If <i>namespace</i> is left blank, the code for the imported schema will be generated in the global namespace.

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/1.0/demos/hello_world`

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 149](#).
 - Derived complex types are not supported, apart from the special case of SOAP arrays.
-

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">
```

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

```
        <input message="tns:greetMe" name="greetMe" />
        <output message="tns:greetMeResponse"
                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, `wSDLtoCcpp`, 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");
}

HelloWorldImplFactory global_HelloWorldImplFactory;

HelloWorldImplFactory::HelloWorldImplFactory()
{
    m_wsdl_location = IT_Bus::String("HelloWorld.wsdl");
    IT_Bus::QName service_name("", "HelloWorldService",
"http://xmlbus.com/HelloWorld");
    IT_Bus::Bus::register_server_factory(
        service_name,
        this
    );
}

HelloWorldImplFactory::~HelloWorldImplFactory()

```

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

```

{
    IT_Bus::QName service_name("", "HelloWorldService",
    "http://xmlbus.com/HelloWorld");
    IT_Bus::Bus::deregister_server_factory(service_name);
    //cleanup();
}

IT_Bus::ServerStubBase*
HelloWorldImplFactory::create_server(IT_Bus::Bus_ptr bus,
IT_Bus::Port* port)
{
    return new HelloWorldImpl(bus, port);
}

const IT_Bus::String &
HelloWorldImplFactory::get_wsdl_location()
{
    return m_wsdl_location;
}

void
HelloWorldImplFactory::destroy_server(IT_Bus::ServerStubBase*
server)
{
    if (server != 0)
    {
        delete IT_DYNAMIC_CAST(HelloWorldImpl*, 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.

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 3 on page 11](#) shows how the `main()` for the server defined by the Converter contract might look.

Example 3: *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::init(argc, argv);

        IT_Bus::run();
    }
    catch (IT_Bus::Exception& e)
    {
        cout << "Error occurred: " << e.Error() << 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 three constructors as shown in [Example 4 on page 12](#).

Example 4: *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);
```

No argument constructor

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.

One argument constructor

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.

Three argument constructor

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 5 on page 13](#), you could instantiate the client proxy to use either HTTP or Tuxedo.

Example 5: *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");
```

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 shutdown before exiting.

Full client code

A client developed to access the service defined by the `HelloWorldService` contract will look similar to [Example 6](#).

Example 6: *HelloWorld Client*

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

IT_USING_NAMESPACE_STD
using namespace IT_Bus;

using namespace HW;
```

Example 6: *HelloWorld Client*

```

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`.

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
```

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:

Operations and Parameters	page 22
Exceptions	page 26
Memory Management	page 33
Implementing a Server Factory	page 46
Multi-Threading	page 51

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.

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

You can declare a WSDL operation 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 7 on page 22](#)).
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 7 on page 22](#)).
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 7](#) shows an example of a simple operation, `testParams`, which takes two input parameters, `inInt` and `inoutInt`, and two output parameters, `inoutInt` and `outFloat`.

Example 7: WSDL Declaration of the testParams Operation

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

Example 7: WSDL Declaration of the *testParams* Operation

```

<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 8](#) shows how the preceding WSDL *testParams* operation (from [Example 7 on page 22](#)) maps to C++.

Example 8: 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 9](#) shows an example of an operation, `testReverseParams`, whose parameters are listed in the opposite order to that of the preceding `testParams` operation.

Example 9: 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>
    ...
  </portType>
</definitions>
```

C++ mapping of testReverseParams

[Example 10](#) shows how the preceding WSDL `testReverseParams` operation (from [Example 9](#) on page 24) maps to C++.

Example 10: C++ Mapping of the testReverseParams Operation

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

Order of in, inout and out parameters

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

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

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:

Non-Propagating Exceptions	page 27
Propagating Exceptions	page 29

Non-Propagating 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 generated exceptions with two 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;
```

`IT_Bus::Exception::Error()`

`Error()` returns an error code, if one is assigned to the exception, that identifies the exception. It has the following signature:

```
IT_ULong Error() const;
```

Currently only the following exceptions have been given error codes:

Table 2: *Artix Exception Error Codes*

Error Code	Description
<code>IT_HTTP_E_COMM_ERROR</code>	A communication error occurred.
<code>IT_HTTP_E_ACCESS_DENIED</code>	Username or password validation error by the server.
<code>IT_HTTP_E_BAD_CONFIG</code>	The configuration file is not valid.
<code>IT_HTTP_E_NOT_FOUND</code>	The URL or file was not found.
<code>IT_HTTP_E_SHUTTING_DOWN</code>	The system is entering a quiescent state.
<code>IT_BUS_E_FAULT</code>	A SOAP fault was returned by the server.

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 is a route is improperly defined. It is defined in `<it_bus/invalid_route_exception.h>`.

Propagating Exceptions

Overview

Artix servers propagate certain exceptions, such as serialization and deserialization exceptions, back to their clients so the client can handle the error gracefully. This is done using the `IT_Bus::FaultException` class, defined in `<it_bus/fault_exception.h>`. `FaultException` extends `Exception` to provide connection awareness and serialization.

Artix propagates user-defined exceptions back to client processes. To specify that an exception is to be propagated, 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.

Declaring a fault in WSDL

[Example 11](#) 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 11: Declaration of the `SampleFault` Fault

```

1 <?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/">
2      <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"/>

```

Example 11: *Declaration of the SampleFault Fault*

3

```

        <output message="tns:echoIntegerResponse"
              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>` tag must be added to the scope of the operation (or operations) which can raise this particular type of fault.

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

Raising a fault exception in a server

[Example 12](#) 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 12: *Raising the SampleFault Fault in the Server*

```

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

```

Example 12: *Raising the SampleFault Fault in the Server*

```

        IT_THROW_DECL((IT_Bus::Exception))
    {
        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 13](#) 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 13: *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 13: *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 34
Assignment and Copying	page 39
Deallocating	page 41
Smart Pointers	page 42

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 42](#) 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 14](#) shows an example of a WSDL operation, `testSeqParams`, with three parameters, `inSeq`, `inoutSeq`, and `outSeq`, of sequence type, `xsd:SequenceType`.

Example 14: 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 14: 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 15](#) shows how to allocate, initialize, and deallocate parameters when calling the `testSeqParams` operation.

Example 15: 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 15: *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 16](#) shows how the parameters are used on the server side, in the C++ implementation of the `testSeqParams` operation.

Example 16: *Server Calling the testSeqParams Operation*

```

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

```


Example 16: *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 39](#) 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 14 on page 34](#) 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 42](#).

Recursive deallocation

The Artix C++ types are designed to support recursive deallocation. That is, if you have an instance, τ , 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 τ . 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.](#)
 - [Assignment semantics.](#)
 - [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.

Assignment semantics

The `auto_ptr` smart pointer types have destructive copy semantics. For example, consider the following assignment between smart pointers of `SequenceTypePtr` type:

```
// C++
SequenceTypePtr assign_from = new SequenceType();
// Initialize assign_from (not shown) ...

SequenceTypePtr assign_to = new SequenceType();
// Initialize assign_to (not shown) ...

// Assignment Statement
assign_to = assign_from;
```

After the assignment, the following facts hold:

- `assign_to` now owns the data previously owned by `assign_from`.
- `assign_from` is reset to a nil pointer (equals 0).
- The data previously owned by `assign_to` has been deleted.

Note: If you are familiar with the CORBA IDL-to-C++ mapping, you should note that these assignment semantics are different from the CORBA `_var` types' assignment semantics.

Client example using simple pointers

[Example 17](#) shows how to call the `testSeqParams` operation using parameters that are allocated on the heap and referenced by *simple pointers*

Example 17: 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();
```

Example 17: *Client Calling testSeqParams Using Simple Pointers*

```

// 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 18](#) shows how to call the `testSeqParams` operation using parameters that are allocated on the heap and referenced by *smart pointers*

Example 18: *Client Calling testSeqParams Using Smart Pointers*

```

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

    BaseClient bc;

    // Allocate all parameters

```


Example 18: *Client Calling testSeqParams Using Smart Pointers*

```

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.

Implementing a Server Factory

Overview

A server factory is responsible for managing the lifecycle of servant objects. Although the WSDL-to-C++ compiler can provide a convenient default implementation of the server factory class, in a realistic application you would typically need to customize the default.

Server factory features

By writing a custom server factory implementation, you can exploit the following features of the server factory design:

- Override the WSDL location.
 - Register a server factory against multiple services.
 - Register multiple ports per service.
 - Create multiple servants per port or share one servant between ports.
-

Default server factory

When you run the `wsdltocpp` utility with the `-impl` flag, it generates a default implementation of a servant class and a server factory class in the files `PortTypeImpl.h` and `PortTypeImpl.cxx`.

The default server factory, generated by `wsdltocpp`, has the following general characteristics:

- A global static instance of the server factory is declared in the `PortTypeImpl.cxx` file.
- The server factory registers itself against a single service and a single port (as specified by the `-e` and `-t` parameters of `wsdltocpp`).
- The threading model defaults to `MULTI_INSTANCE`.

Sample WSDL

[Example 19](#) shows an extract from a WSDL contract that defines multiple services and ports for the `HelloWorld` port type. The `SOAPHelloWorldService` service defines a single port that exposes `HelloWorld` as a SOAP service and the `HW>HelloWorldService` service defines two ports that expose `HelloWorld` as a CORBA service.

Example 19: Sample WSDL with Multiple Services and Ports

```
<definitions ... >
  ...
  <service name="SOAPHelloWorldService">
    <port binding="tns:SOAPHelloWorldPortBinding"
          name="SOAPHelloWorldPort">
      <soap:address location="http://localhost:8080"/>
    </port>
  </service>
  <service name="HW>HelloWorldService">
    <port name="HW>HelloWorldPort"
          binding="tns:HW>HelloWorldBinding">
      <corba:address location="file://../HelloWorld.ior"/>
    </port>
    <port name="HW.ALTHelloWorldPort"
          binding="tns:HW>HelloWorldBinding">
      <corba:address
        location="corbaname:rir:/NameService#helloWorld"/>
    </port>
  </service>
</definitions>
```

Server factory example

[Example 20](#) shows an example of a server factory class that is customized to register multiple services and ports. This server factory implementation is based on the WSDL contract from [Example 19 on page 47](#).

Example 20: Example Implementation of a Server Factory Class

```
// C++
...
1 HW>HelloWorldImplFactory global_HW>HelloWorldImplFactory;

HW>HelloWorldImplFactory::HW>HelloWorldImplFactory()
{
    m_wsdl_location = IT_Bus::String("HelloWorld.wsdl");
}
```

Example 20: *Example Implementation of a Server Factory Class*

```

2   IT_Bus::QName service_nameSOAP("", "SOAPHelloWorldService",
   "http://schemas.iona.com/idl/HelloWorld.idl");
   IT_Bus::Bus::register_server_factory(
       service_nameSOAP,
       this
   );

   IT_Bus::QName service_name("", "HW.HelloWorldService",
3   "http://schemas.iona.com/idl/HelloWorld.idl");
   IT_Bus::Bus::register_server_factory(
       service_name,
       this,
       "HW_HelloWorldPort"
   );
4   IT_Bus::Bus::register_server_factory(
       service_name,
       this,
       "HW_ALTHelloWorldPort"
   );

}

HW_HelloWorldImplFactory::~HW_HelloWorldImplFactory()
{
   IT_Bus::QName service_name("", "HW.HelloWorldService",
5   "http://schemas.iona.com/idl/HelloWorld.idl");
   IT_Bus::Bus::deregister_server_factory(service_name);
}

6 IT_Bus::ServerStubBase* HW_HelloWorldImplFactory::create_server(
   IT_Bus::Bus_ptr bus, IT_Bus::Port* port)
{
   return new HW_HelloWorldImpl(bus, port);
}

const IT_Bus::String &
7 HW_HelloWorldImplFactory::get_wsdl_location()
{
   return m_wsdl_location;
}

IT_Bus::ThreadingModel
8 HW_HelloWorldImplFactory::get_threading_model() const
{

```

Example 20: *Example Implementation of a Server Factory Class*

```

    return IT_Bus::MULTI_INSTANCE;
}

9 void HW>HelloWorldImplFactory::destroy_server(
    IT_Bus::ServerStubBase* server
)
{
    if (server != 0)
    {
        delete IT_DYNAMIC_CAST(HW>HelloWorldImpl*, server);
    }
}

```

The preceding server factory example can be explained as follows:

1. This line creates a global static instance of the server factory, which is the default way of creating the server factory. This approach is not mandatory, however. You could delete this line and create the server factory instance at a different point in the server code.
2. The constructor is the usual place where a server factory registers itself against particular services and ports. This line calls `IT_Bus::Bus::register_server_factory()` to register the server factory against the `SOAPHelloWorldService` service.
3. This line registers the server factory against the `HW>HelloWorldService` service (CORBA service) and the `HW>HelloWorldPort` port. Note that this form of `register_server_factory()` explicitly specifies the port name.
4. This line registers the server factory against the `HW>HelloWorldService` service (CORBA service) and the `HW>ALHelloWorldPort` port.
5. You can deregister services in the server factory destructor.
6. The `create_server()` function is called by Artix whenever a servant instance (of `HW>HelloWorldImpl` type) is needed. The pattern of calls to `create_server()` is affected by the current threading model—see [“Server Threading Models” on page 54](#).
7. The `get_wsdl_location()` function is called by Artix to find the WSDL contract to use with this server factory. By changing the return value of this function, you can direct Artix to look for a different WSDL contract to use with the server factory.

8. The `get_threading_model()` function is called by Artix to determine the threading model to use with this server factory. For more details, see [“Server Threading Models” on page 54](#).
9. The `destroy_server()` function is called by Artix to clean up servant instances.

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 52
Server Threading Models	page 54
Changing the Server Threading Model	page 58

Client Threading Issues

Client threading

The client proxy classes and the runtime library are thread-safe, in that multi-threaded applications may safely use the library from multiple threads simultaneously. However, a single client proxy instance should not be shared among multiple threads without serializing access to the instance.

Single client proxy in two threads

Example 21 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 21: *Single Client Proxy in Two Threads*

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

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

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 22](#) 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 22: *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();
}
```

Server Threading Models

Overview

Artix support 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 [“Changing the Server Threading Model”](#) on page 58). This subsection provides an overview of each of the server-side threading models in Artix, as follows:

- [MULTI_INSTANCE](#).
- [MULTI_THREADED](#).

MULTI_INSTANCE

The `MULTI_INSTANCE` 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 1](#) shows an outline of the `MULTI_INSTANCE` 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.

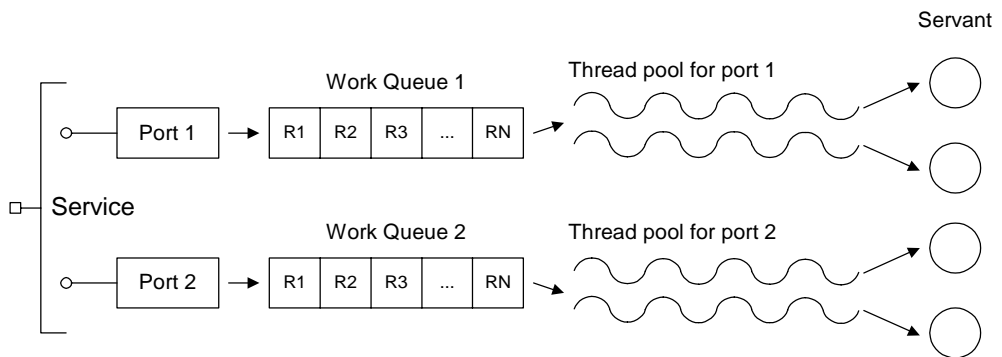


Figure 1: Outline of the `MULTI_INSTANCE` Threading Model

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

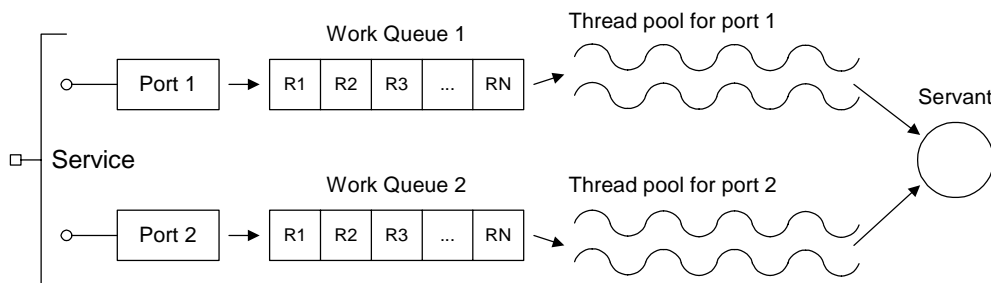


Figure 2: Outline of the `MULTI_THREADED` Threading Model

Default threading model

The default threading model is `IT_Bus::MULTI_INSTANCE`.

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 23](#) can be used to configure thread pools at the global level; that is, these settings would apply to all services by default.

Example 23: 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:ServiceName:initial_threads
thread_pool:ServiceName:low_water_mark
thread_pool:ServiceName:high_water_mark
```

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 24](#) show how to configure the thread pool for a service named `SessionManager`.

Example 24: *Thread Pool Settings at the Service Name Level*

```
# Artix configuration file

sample_config {
    ...
    # Thread pool settings at Service name level
    thread_pool:SessionManager:initial_threads = "1";
    thread_pool:SessionManager:low_water_mark = "5";
    thread_pool:SessionManager:high_water_mark = "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:NamespaceURI:ServiceName:initial_threads
thread_pool:NamespaceURI:ServiceName:low_water_mark
thread_pool:NamespaceURI:ServiceName:high_water_mark
```

Where *NamespaceURI* is the namespace URI in which *ServiceName* is defined.

For example, the settings in [Example 25](#) show how to configure the thread pool for a service named `SessionManager` in the `//my.tns1/` namespace URI.

Example 25: *Thread Pool Settings at the Qualified Service Name Level*

```
# Artix configuration file

sample_config {
    ...
    # Thread pool settings at Service name level
    thread_pool:http://my.tns1/:SessionManager:initial_threads =
    "1";
    thread_pool:http://my.tns1/:SessionManager:low_water_mark =
    "5";
    thread_pool:http://my.tns1/:SessionManager:high_water_mark =
    "10";
};
```

Changing the Server Threading Model

Overview

This subsection explains how to change the server threading model by programming. The server threading model can be specified on a per-service basis.

Threading model options

The `it_bus/threading_model.h` header file defines the following threading model options, as shown in [Example 26](#).

Example 26: Threading Model Options

```
namespace IT_Bus
{
    enum ThreadingModel
    {
        MULTI_INSTANCE = 0,
        MULTI_THREADED = 1,
        SINGLE_THREADED = 2
    };
};
```

ServerFactoryBase class

The `ServerFactoryBase` class, as shown in [Example 27](#), defines the server factory API. All of the member functions are abstract, except for `get_threading_model()`, which has a default implementation that returns `IT_Bus::MULTI_INSTANCE`.

Example 27: The ServerFactoryBase Class

```
// C++
class IT_BUS_API ServerFactoryBase
{
public:
    ServerFactoryBase();
    virtual ~ServerFactoryBase();

    virtual ServerStubBase*
    create_server(Bus_ptr bus, Port* port) = 0;

    virtual const String & get_wsdl_location() = 0;

    virtual void destroy_server(ServerStubBase* server) = 0;
```

Example 27: *The ServerFactoryBase Class*

```
virtual ThreadingModel
get_threading_model() const;
};
```

get_threading_model() function

Artix calls the `get_threading_model()` function at start-up time to determine which threading model to use for this service. You can change the threading model by returning a non-default value from this function.

create_server() function

Artix calls the `create_service()` function whenever a new service instance is needed. The pattern of `create_server()` calls depends on the chosen threading model, as described in [Table 3](#).

Table 3: *Pattern of create_server() Calls in Various Threading Models*

Threading Model	Pattern of create_server() Calls
MULTI_INSTANCE	<code>create_server()</code> is called once for each thread in the thread pool (see “Thread pool configuration levels” on page 56).
MULTI_THREADED	<code>create_server()</code> is called once only.

Overriding `get_threading_model()`

To change the threading model for a particular service, you should override the default implementation of `get_threading_model()`.

For example, if you have a service of `HelloWorld` port type, the `wsdltocpp` generates a default implementation of the server factory, `HelloWorldImplFactory`, in the files `HelloWorldImpl.h` and `HelloWorldImpl.cxx`. To change the threading model to `MULTI_THREADED` in this case, perform the following steps:

1. Edit the `HelloWorldImpl.h` file, adding a declaration of the `get_threading_model()` function to the `HelloWorldImplFactory` class:

```
// C++
class HelloWorldImplFactory : public
    IT_Bus::ServerFactoryBase
{
public:
    ...
    virtual ThreadingModel get_threading_model() const;
};
```

2. Edit the `HelloWorldImpl.cxx` file, adding an implementation of the `get_threading_model()` function as follows:

```
// C++
IT_Bus::ThreadingModel
HelloWorldImplFactory::get_threading_model() const
{
    return IT_Bus::MULTI_THREADED;
}
```


Artix References

An Artix reference is a handle to a particular port on a particular service. 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 62
The IT_Bus::Reference Class	page 65
Using the Artix Locator	page 66

Introduction to References

Overview

An Artix reference encapsulates the location information for a particular WSDL port on a particular WSDL service. When compared with storing location information in WSDL, references have the following advantages:

- References are more dynamic—that is, the information encapsulated in a reference is only partially dependent on the WSDL contract. Hence, reference details can change at runtime.
- References can be sent across the wire as parameters of or return values from operations.
- References can be stored in a central repository, facilitating features such as load balancing and directory enquiries.

Contents of an Artix reference

An Artix reference encapsulates the following data:

- *Binding QName*—the qualified name of the binding with which this reference is associated.
- *Service QName*—the qualified name of the service with which this reference is associated.
- *Port name*—identifies the port with which this reference is associated.
- *WSDL location*—this data is included only as a backup, in case the client does not already have access to the WSDL contract. In most cases, the client would already have a local or cached copy of the WSDL contract.
- *Properties*—a list of opaque properties, which makes the reference type arbitrarily extensible. The properties list is typically used to hold binding-specific data. In the future, properties might be used to flag particular qualities of service as well.

XML representation of a reference

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

ArtixInstallDir/artix/Version/schemas/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. For details of the `IT_Bus::Reference` API, see [“The `IT_Bus::Reference` Class” on page 65](#).

Static references

A *static reference* is a reference for which all of the port and service details appear explicitly in the WSDL contract. The static reference, therefore, delegates most of the details to the WSDL contract. [Figure 3](#) illustrates the relationship between a static reference and the WSDL contract.

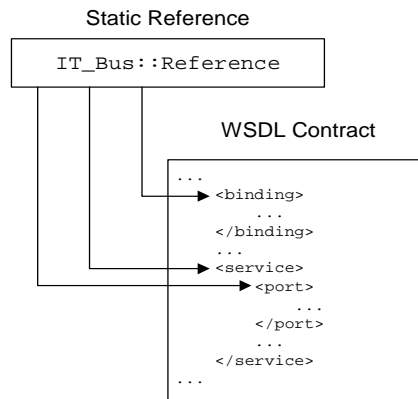


Figure 3: A Static Reference

The typical contents of a static reference are as follows:

- *Binding QName*—a particular binding in the WSDL contract.
- *Service QName*—a particular service in the WSDL contract.
- *Port name*—the name of one of the ports, from the preceding service, in the WSDL contract.
- *WSDL location*—the location of a backup copy of the WSDL contract.
- *Properties*—not required for static references. However, Artix normally caches port addressing information in the properties. This optimization can help clients to avoid parsing the WSDL contract, as long as the client has already parsed the relevant binding.

Transient references

A *transient reference* stores all of its service and port attributes explicitly in a properties list, rather than referring to the WSDL contract. Hence, a transient reference is more flexible, because it can refer to endpoints created at runtime. Figure 4 illustrates the relationship between a transient reference and the WSDL contract.

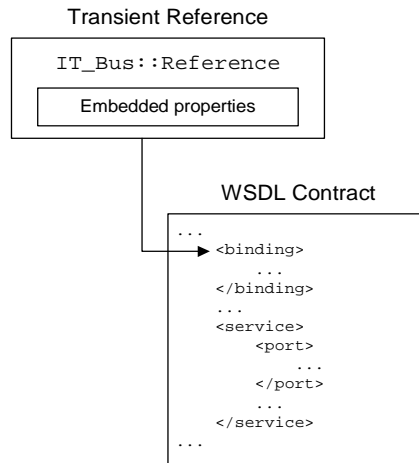


Figure 4: A Transient Reference

The typical contents of a transient reference are as follows:

- *Binding QName*—a particular binding in the WSDL contract.
- *WSDL location*—the location of a backup copy of the WSDL contract.
- *Properties*—contains the kind of data you would normally find in a `<service>` tag and a `<port>` tag. This data is binding-specific and it enables the client to open a connection to an endpoint on the server.

The service QName and port name are *not* used by transient references; they are initialized as empty strings.

The IT_Bus::Reference Class

Overview

The `IT_Bus::Reference` class provides the following kinds of member function:

- [Setting and getting basic reference properties.](#)
- [Getting binding-specific properties.](#)

Setting and getting basic reference properties

The following `IT_Bus::Reference` member functions enable you to get and set a reference's service QName, port name, binding QName and WSDL file location:

```
IT_Bus::QName & get_service_qname();  
IT_Bus::String & get_port_name();  
  
IT_Bus::QName & get_binding_qname();  
IT_Bus::String & get_wsdll_location();  
  
void set_service_qname(const IT_Bus::QName & service_qname);  
void set_port_name(const IT_Bus::String & port_name);  
  
void set_binding_qname(const IT_Bus::QName & binding_qname);  
void set_wsdll_location(const IT_Bus::String & wsdl_location);
```

Getting binding-specific properties

The following `IT_Bus::Reference` member function enables you to get a list of binding-specific properties:

```
IT_Bus::AnyHolderList & get_properties();
```

These binding-specific properties are usually needed only for transient references. The properties are read and interpreted by the relevant binding plug-in. Hence, you would not normally need to access these properties in your application code.

Using the Artix Locator

Overview

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 section

This section contains the following subsections:

Overview of the Locator	page 67
Locator WSDL	page 69
Registering Endpoints with the Locator	page 75
Reading a Reference from the Locator	page 76
Pausing and Resuming Endpoints	page 80

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 ports are registered against a single service QName, the locator load balances over the service's ports using a round-robin algorithm.

Figure 5 gives a general overview of the locator architecture.

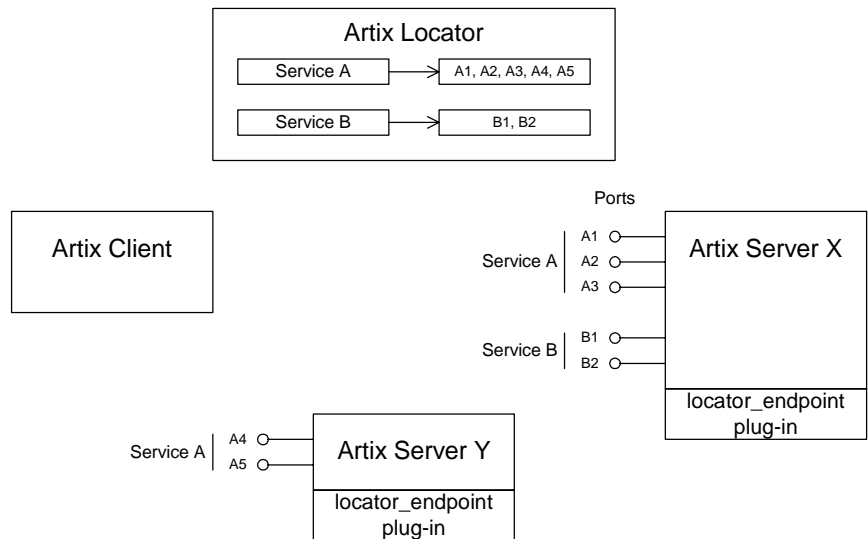


Figure 5: *Artix Locator Overview*

Locator demonstration

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

`ArtixInstallDir/artix/Version/demos/locator`

Locator service

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

- *Standalone deployment*—the locator is deployed as an independent server process (as shown in [Figure 5](#)). This approach is described in detail in the “Using the Artix Locator Service” chapter from the *Artix User’s Guide*. Sample source code for such a standalone locator service is provided in the `demos/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.
-

Registering endpoints

An Artix *endpoint* is a particular WSDL port in a particular WSDL service. A server registers its endpoints (that is, WSDL ports) 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 ports are registered against a single service QName in the locator, the locator will employ a round-robin algorithm to pick one of the ports. Hence, the locator effectively *load balances* a service over all of its registered ports.

For example, [Figure 5 on page 67](#) shows the `Service A` service with five ports registered against it, `A1`, `A2`, `A3`, `A4`, and `A5`. When the Artix client looks up a reference for `Service A`, it obtains an Artix reference, `Ax`, 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. 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.

Note: Currently, the locator service is limited by the fact that most Artix bindings do not support endpoint references. In future releases of Artix, when the support for references is extended to other bindings, it should be possible to use the locator with other bindings and transports.

WSDL contract

[Example 28](#) 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 28: *Extract from the Locator WSDL Contract*

```

1 <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 targetNamespace="http://ws.iona.com/locator">
      <xs:import
        schemaLocation="../../schemas/references.xsd"
        namespace="http://schemas.iona.com/references"/>
      ...
    
```

Example 28: *Extract from the Locator WSDL Contract*

```

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">
    <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" />
      </operation>
    </portType>
</binding>
</service>

```

Example 28: *Extract from the Locator WSDL Contract*

```

        message="ls:endpointNotExistFault" />
    </operation>
</portType>
<binding name="LocatorServiceBinding"
    type="ls:LocatorService">
    ...
</binding>
<service name="LocatorService">
    <port name="LocatorServicePort"
        binding="ls:LocatorServiceBinding">
        <soap:address
7 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 schema definition of the `ref:Reference` type. You might have to edit the value of the `schemaLocation` attribute, if the `references.xsd` schema file is stored in a different location relative to the `locator.wsdl` file.
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.

Note: Currently, it is not possible to specify a particular port in the lookup query.

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 port is registered against a particular service name, the locator picks one of the ports 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, which is called by Artix clients to retrieve endpoint references, is the only operation from the `LocatorService` port type that user applications would typically need.
7. The SOAP `location` attribute specifies the host and IP port for the locator service. If you want the locator to run on a different host and listen on a different IP port, you should edit this setting.

C++ mapping

Example 29 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 29: 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>

namespace IT_Bus_Services
{
    class LocatorServiceClient : public LocatorService, public
    IT_Bus::ClientProxyBase
    {
    private:

    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,
```

Example 29: C++ Mapping of the LocatorService Port Type

```

        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_Services::lookupEndpoint &
                                     lookupEndpoint_in,
    IT_Bus_Services::lookupEndpointResponse &
                                     lookupEndpointResponse_out
) IT_THROW_DECL((IT_Bus::Exception));
};
};
};

```

The lookupEndpoint type

The input parameter for the `lookup_endpoint` operation is of `lookupEndpoint` type, which maps to C++ as follows:

```

// C++
namespace IT_Bus_Services
{
    class lookupEndpoint : public IT_Bus::SequenceComplexType
    {
    public:
        lookupEndpoint();
        lookupEndpoint(const lookupEndpoint& copy);
        virtual ~lookupEndpoint();

        const IT_Bus::QName & getservice_qname() const;
        IT_Bus::QName & getservice_qname();
        void setservice_qname(const IT_Bus::QName & val);
        ...
    };
};
};

```

The lookupEndpointResponse type

The output parameter for the `lookup_endpoint` operation is of `lookupEndpointResponse` type, which maps to C++ as follows:

```
// C++
namespace IT_Bus_Services
{
    class lookupEndpointResponse
    : public IT_Bus::SequenceComplexType
    {
    public:
        lookupEndpointResponse();
        lookupEndpointResponse(const lookupEndpointResponse&
copy);
        virtual ~lookupEndpointResponse();
        ...
        const IT_Bus::Reference & getservice_endpoint() const;
        IT_Bus::Reference & getservice_endpoint();
        void setservice_endpoint(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. Once the appropriate plug-ins are loaded, the server will automatically register every endpoint (that is, service/port combination) that is created on the server side.

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 must be configured to include the `soap`, `http`, and `locator_endpoint` plug-ins, as shown in the following `demo.locator.server` configuration scope from `artix.cfg`:

```
# Artix Configuration File (artix.cfg)
...
demo {
  locator {
    server
    {
      plugins:locator:wSDL_url="../wSDL/locator.wSDL";
      orb_plugins = ["xmlfile_log_stream", "iiop_profile",
"giop", "iiop", "soap", "http", "tunnel", "ots", "fixed",
"ws_orb", "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
```

References

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

- “Using the Artix Locator Service” chapter from the *Artix User’s Guide*.
- The Artix `locator` demonstration in `artix/Version/demos/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 6](#) shows an outline of how a client bootstraps a connection in this way.

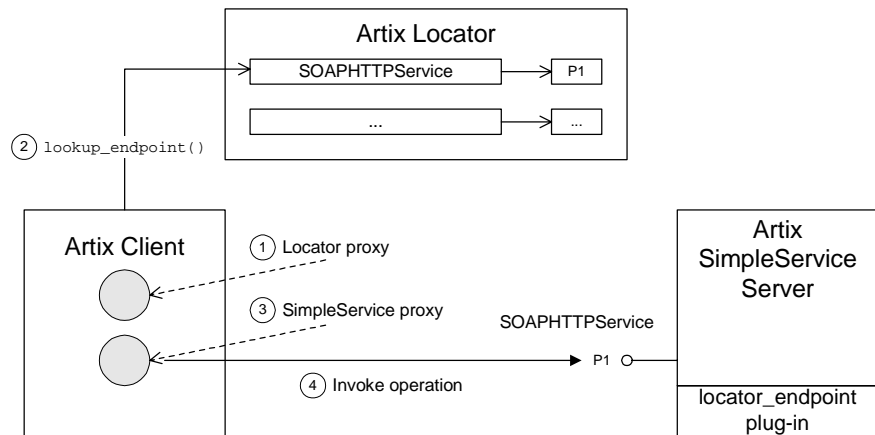


Figure 6: Steps to Read a Reference from the Locator

Programming steps

The main programming steps needed to read a reference from the locator, as shown in [Figure 6](#), 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 30 shows an example of the code for an Artix client that retrieves a reference to a `SimpleService` service from the Artix locator.

Example 30: *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;
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::init(my_argc, (char **)my_argv);

2         QName service_name(
            "", "LocatorService", "http://ws.iona.com/locator"
        );
3         QName sh_service_name(
            "", "SOAPHTTPService", "http://www.iona.com/bus/tests"
        );

4         String port_name("LocatorServicePort");

        // 1. Construct a locator service proxy
5         IT_Bus_Services::LocatorServiceClient*
            m_locator_client = new LocatorServiceClient(
                "../wsdl/locator.wsdl", service_name, port_name

```

Example 30: *Example of Reading a Reference from the Locator Service*

```

        );

        // Setup input and output parameters to locator
        lookupEndpoint sh_input;
        sh_input.setservice_qname(sh_service_name);
        lookupEndpointResponse sh_output;

        // 2. Invoke on locator
6      m_locator_client->lookup_endpoint(
            sh_input,
            sh_output
        );

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

        // 4. Use your new proxy
8      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, `service_name`, that identifies the `<service name="LocatorService">` tag from the locator WSDL. See the listing of the locator WSDL in [Example 28 on page 69](#).
3. This line constructs a qualified name, `sh_service_name`, that identifies the `SOAPHTTPService` service from the `SimpleService` WSDL.
4. This port name refers to the `<port name="LocatorServicePort" ...>` tag in the locator WSDL (see [Example 28 on page 69](#)).
5. The locator service proxy is created by calling the three-argument constructor for the `LocatorServiceClient` class. The three arguments passed (locator WSDL, service name, and port name) specify the locator endpoint exactly.
6. 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.

7. The call to `sh_output.getservice_endpoint()` extracts the returned `SimpleService` reference which is then passed to a simple client proxy constructor. The constructor is a special form that takes an `IT_Bus::Reference` type as its argument:

```
// C++
SimpleClient(
    IT_Bus::Reference & reference,
    IT_Bus::Bus_ptr bus = 0
);
```

8. You can now use the simple client proxy to make invocations on the remote Artix server.

Pausing and Resuming Endpoints

Overview

As part of a load management strategy, it is useful if you can pause the traffic of requests incoming to a server. For this purpose, the `IT_Bus::Service` class provides a pair of functions to pause and resume a service's endpoints. The `locator_endpoint` plug-in supports this functionality by de-registering the service's endpoints from the locator. This does not prevent existing clients from sending requests to the server, but it does help to limit the load by making the server temporarily unavailable to new clients.

IT_Bus::Service pause and resume functions

The `IT_Bus::Service` class provides the following member functions for pausing and resuming an Artix service:

IT_Bus::Service::reached_capacity()

Call the `reached_capacity()` function to pause a service's endpoints. The `locator_endpoint` plug-in listens for this event and, when the function is called, the `locator_endpoint` plug-in deregisters the service's endpoints (ports) from the locator.

IT_Bus::Service::below_capacity()

Call the `below_capacity()` function to resume a service's endpoints. The `locator_endpoint` plug-in listens for this event and, when the function is called, the `locator_endpoint` plug-in re-registers the service's endpoints with the locator.

C++ server example

[Example 31](#) shows how to pause and resume the endpoints for a BookService service.

Example 31: Code to Pause and Resume a Service's Endpoints

```
// C++
// Get handle to Service from Bus if available
IT_Bus::QName service_name("", "BookService", "http://books");
IT_Bus::Service* = bus->get_service(service_name);

// Trigger the de-register if registered
service->reached_capacity();
...
// Trigger the re-register if not register
service->below_capacity();
```


Using Sessions in Artix

The Artix Session Manager helps you manage service resources.

In this chapter

This chapter discusses the following topics:

Introduction to Session Management in Artix	page 84
Registering a Server with the Session Manager	page 87
Working with Sessions	page 90

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 7 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.

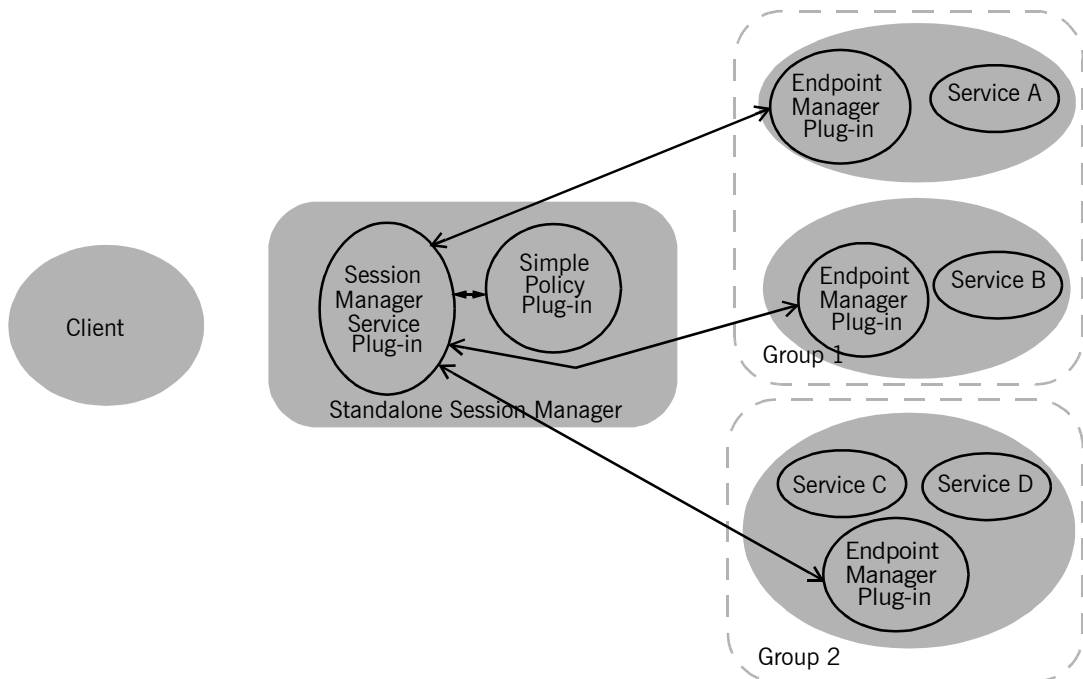


Figure 7: *The Session Manager Plug-ins*

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 7, 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 water-slide group, it would ask the session manager for a session with the water-slide group. The session manager would then check and see if the water-slide group had an available session, and if so it would return a session id and the list of water-slide service references to the client. The session manager would then notify the endpoint managers in the water-slide 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 water-slide 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 water-slide 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`
- `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:

`plugins:session_endpoint_manager:wSDL_url` points to the contract describing the contact information for the session manager that will be managing the services.

`plugins:session_endpoint_manager:endpoint_manager_url` points to the contract describing the contact information for the endpoint manager for this server. This enables the session manager to contact the service to with updated state information.

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

[Example 32](#) shows the configuration scope of a server that hosts services managed by the session manager.

Example 32: *Server Configuration Scope*

```
qajaq_server
{
  orb_plugins = ["xmlfile_log_stream", "soap", "http", "fixed", "session_endpoint_manager"];
  plugins:session_endpoint_manager:wSDL_url="session-manager-service.wsdl";
  plugins:session_endpoint_manager:endpoint_manager_url="session-manager-endpoint.wsdl";
  plugins:session_endpoint_manager:default_group="qajaq_group";
};
```

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

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

Example 33: *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 `plugins:session_endpoint_manager:endpoint_manager_url` 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 `plugins:session_endpoint_manager:wsdl_url`.

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.

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 34](#) shows how to instantiate a session manager proxy.

Example 34: *Instantiating a Session Manager Proxy*

```
// C++
SessionManagerClient session_manager_proxy = new
    SessionManagerClient("session_manager.wsdl");
```

For more information on instantiating Artix proxies, see the *Artix C++ Programmer's Guide*.

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:

```
void begin_session(IT_Bus_Services::BeginSession input,  
                  IT_Bus_Services::BeginSessionResponse output);
```

`input` contains the name of the desired group and the desired duration of the session. The group name is set using the `setendpoint_group()` method. The group name can be any valid string and corresponds to the default group name set in the service's configuration scope as described in ["Configuring the server" on page 87](#).

The session duration is set using the `setpreferred_renew_timeout()` method. The duration is specified 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.

`output` contains the information needed to use the session.

Once a session is returned in `output`, you will need to extract the session ID to work with the session. This is done using `getsession_id()`.

`getsession_id()` returns the session ID as an

`IT_Bus_Services::SessionID`.

Example 35 shows the client code to begin a session for `qajaq_group`.

Example 35: Beginning a Session

```
// C++
IT_Bus_Services::BeginSession begin_session_request;
IT_Bus_Services::BeginSessionResponse begin_session_response;

// set the group to request
begin_session_request.setendpoint_group("qajaq_group");
// set session renewal interval to 10 mins
begin_session_request.setpreferred_renew_timeout(600);

session_mgr.begin_session(begin_session_request,
                          begin_session_response);

IT_Bus_Services::SessionId session;
session =
    begin_session_response.getsession_info().getsession_id();
```

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:

```
void get_all_endpoints(IT_Bus_Services::GetAllEndpoints request,
                     IT_Bus_Services::GetAllEndpointsResponse response)
```

`request` contains the session ID for which you are requesting services. Set the session ID using the `setsession_id()` method on `request` with the session ID returned from the session manager.

`response` contains the list of services returned from `get_all_endpoints()`. If the group has no services, `response` will be empty.

Example 36 shows how to get the list of services for a group.

Example 36: *Retrieving the List of Services in a Group*

```
//C++
IT_Bus_Services::GetAllEndpoints request;
IT_Bus_Services::GetAllEndpointsResponse response;

// group session initialized above.
get_all_endpoints_request.setsession_id(session);

session_mgr.get_all_endpoints(request, response);
```

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. To instantiate the proxy, you first need to narrow down the list returned services to the desired one. `GetAllEndpointsResponse` contains an array of references to active services that can be retrieved using `GetAllEndpointsResponse`'s `getendpoints()` method. You can use simple indexing to get one of the references. For example, to use the first service in the list you would use the following:

```
response.getendpoints()[0]
```

Because the session manager simply returns the services in the order the services registered with the session manager, the clients must be 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 may

want 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 37](#).

Example 37: *Checking the Service Reference for its Interface*

```
//C++
IT_Bus::Reference endpoint = response.getendpoints()[0];
if (endpoint.get_service_name() ==
    QName("", "QajaqService", "http://qajaqs.com"))
{
    // instantiate a QajaqService using endpoint
}
else
{
    // do something else
}
```

[Example 38](#) shows the client code for creating a proxy `qajaq` server from a group service.

Example 38: *Instantiate a Proxy Server*

```
// C++
QajaqClient qajaq_proxy(response.getendpoints()[0]);
```

Create a session header

Services that are being managed by the session manager will only accept requests that include a valid session header. The session header information is passed to the server as part of the proxy's input message attributes. Creating the session header and putting into the input message attributes takes three steps:

1. [Set](#) the proxy to use input message attributes.
2. [Get](#) a handle to the proxy's input message attributes.
3. [Set](#) the session information into the input message attributes.

Setting the proxy to use input message attributes

Artix client proxies all support a helper method, `get_port()`, that provides access to the port information used by the client to connect the service. One of an Artix proxy's port properties is `use_input_message_attributes`.

Setting this property to `true` tells the bus to ensure the input message attributes are propagated through to the server. [Example 39](#) shows how to set the client proxy port's `use_input_message_attributes` property to `true`.

Example 39: *Use Input Message Attributes*

```
//C++
// Get the proxy's port
IT_Bus::Port proxy_port = qajaq_proxy.get_port();

// set the port property
proxy_port.use_input_attributes(true);
```

Getting a handle to the input message attributes

A pointer to the proxy port's input message attributes is returned by the port's `get_input_message_attributes()` method. [Example 40](#) shows how to get a handle to the input message attributes.

Example 40: *Getting the Input Message Attributes*

```
MessageAttributes& input_attributes =
    proxy_port().get_input_message_attributes();
```

Setting the session information into the input message attributes

There are two attributes that need to be set to include the proper session information in the input message:

SessionName specifies the name the session manager has given this session. The session manager endpoints in the group will also be given this name to validate session header's against. The session name is returned by invoking `getname()` of the session ID of the active session.

SessionGroup specifies the group name for which the session is valid. The session endpoints also use to ensure that the session is for the correct group. The session group is returned by invoking `getendpoint_group()` on the session ID of the active session.

The input message attributes are set using the message attribute handle's `set_string()` method. `set_string()` takes two attributes. The first is a string specifying the name of the attribute being set. The second is the value to be set for the attribute. [Example 41](#) shows how to set the session information in to the input message attributes.

Example 41: *Setting the Input Message Attributes*

```
// C++
input_attributes.set_string("SessionName", session.getname());
input_attributes.set_string("SessionGroup",
                           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 client 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:

```
void renew_session(IT_Bus_Services::RenewSession params,
                  IT_Bus_Services::RenewSessionResponse renewed);
```

`params` contains the session ID of the session being renewed and the duration, in seconds, of the renewal. The session ID is set using `params'` `setsession_id()` method. The renewal duration is set using `params'` `setrenew_timeout()` method.

If the renewal is successful, `renewed` will return containing the duration of the renewal. The returned duration may be different if the requested renewal duration was outside of the configured range for session timeouts.

If the renewal is unsuccessful, an `IT_Bus_Services::renewSessionFaultException` is raised.

[Example 42](#) shows how to end a session.

Example 42: *Ending a Session*

```
//C++
IT_Bus_Services::RenewSession params;
IT_Bus_Services::RenewSessionResponse renewed;
params.setsession_id(session);
params.setrenewal_timeout(600);
try
{
    session_mgr.renew_session(params, renewed);
}
catch (IT_Bus_Services::renewSessionFaultException)
{
    // handle the exception
}
```

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:

```
void end_session(IT_Bus_Services::EndSession params);
```

`params` contains the session ID of the session being ended. The session ID is set using `params`' `setsession_id()` method.

[Example 43](#) shows how to end a session.

Example 43: *Ending a Session*

```
//C++
IT_Bus_Services::EndSession params;
params.setsession_id(session);
session_mgr.end_session(params);
```


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 100
Transaction API	page 102
Client Example	page 104

Introduction to Transactions

Overview

Artix supports a pluggable model of transaction support, which is currently restricted to the CORBA Object Transaction Service (OTS) only and, by default, supports client-side transaction demarcation only. Other transaction services (such as MQ series transactions) will be supported in a future release. The following transaction features are supported by Artix:

- [Client-side transaction support](#).
- [Compatibility with Orbix ASP](#).
- [Pluggable transaction factory](#).

Client-side transaction support

By default, Artix has only client-side support for CORBA OTS-based transactions. Transaction demarcation functions (`begin()`, `commit()` and `rollback()`) can be used on the client side to control transactions that are hosted on a remote CORBA OTS server, as shown in [Figure 8](#).

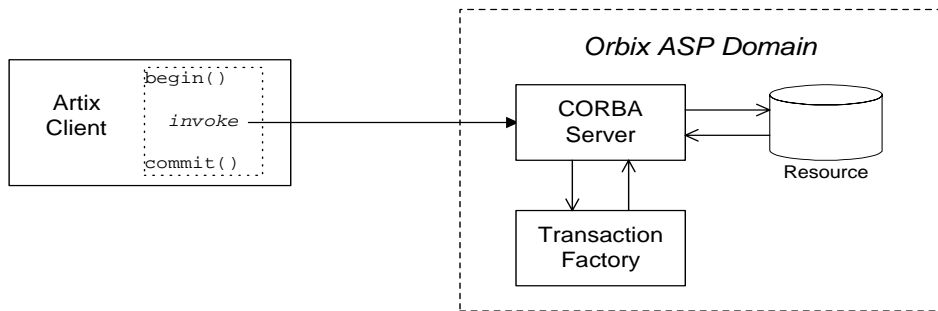


Figure 8: *Artix Client Invokes a Transactional Operation on a CORBA OTS Server*

In [Figure 8](#), the resource and the transaction factory are located on the server side (in an Orbix ASP domain). Artix currently does not have the capability to manage resources on the client side.

Compatibility with Orbix ASP

The Artix transaction facility is fully compatible with CORBA OTS in Orbix ASP. Hence, if you already have a transactional server implemented with Orbix ASP, you can easily integrate this with an Artix client.

Pluggable transaction factory

The underlying transaction factory used by Artix can be replaced within a pluggable framework. In future, Artix will support multiple factories (for example, OTS, MQ series, and so on). Currently, only the following transaction factory is supported:

- `ots`

Transaction API

Overview

The Artix transaction API is provided by the following classes and modules:

- `IT_Bus::Bus`

Note: You can also gain access to interfaces from the `CosTransactions` module, which is part of CORBA OTS, if you have IONA's Orbix ASP product. This is not included with Artix.

IT_Bus::Bus member functions

The `IT_Bus::Bus` class has the following member functions, which are used to manage transactions:

```
// C++
void begin(const char* factory_name);

void commit(bool report_heuristics, const char* factory_name);

void rollback(const char* factory_name);

void rollback_only(const char* factory_name);

char* get_transaction_name(const char* factory_name);

IT_Bus::Boolean within_transaction(const char* factory_name);

void set_timeout(IT_Bus::UInt seconds, const char*
    factory_name);

IT_Bus::UInt get_timeout(const char* factory_name);

CosTransactions::Coordinator*
get_coordinator(const char* factory_name);
```

Factory name parameter

The factory name parameter, which is passed to each of the preceding API functions, identifies the kind of transaction factory that is used. Currently, only the CORBA OTS transaction factory is supported, which is specified by the string, `ots`.

Client transaction functions

The `begin()`, `commit()`, and `rollback()` functions are used to demarcate transactions on the client side. The `commit()` function ends the transaction normally, making any changes permanent. The `rollback()` function aborts the transaction, rolling back any changes.

The `within_transaction()` function, which can be called in an execution context on the server side, returns `TRUE` if the current operation is executing within a transaction scope.

Server transaction functions

The `rollback_only()` function can be called on the server side to mark the current transaction for rollback. After this function is called, the current transaction can only be rolled back, not committed.

Timeouts

A client can use the `set_timeout()` function to impose a timeout on the transactions it initiates. If the timeout is exceeded, the transaction is automatically rolled back.

CosTransactions::Coordinator class

The `CosTransactions::Coordinator` class enables you to exercise fine-grained control over a transaction. The `CosTransactions::Coordinator` class is defined by the CORBA Object Transaction Service (OTS).

Client Example

Overview

This section describes a transactional Artix client that connects to a remote CORBA OTS server. The client uses the Artix transactional API to delimit transactions, where the transactional resource and the transaction factory are both located in the CORBA OTS server. This simple Artix client cannot manage a transactional resource on its own.

WSDL sample

[Example 44](#) defines a WSDL port type, `AccountPortType`, with two operations `withdraw` and `deposit`, which are used for withdrawing money from or depositing money into personal accounts on the server.

Example 44: Definition of an `AccountPortType` Port

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >
  <message name="withdraw">
    <part name="accName" type="xsd:string" />
    <part name="amount" type="xsd:int" />
  </message>
  <message name="withdrawResponse" />
  <message name="deposit">
    <part name="accName" type="xsd:string" />
    <part name="amount" type="xsd:int" />
  </message>
  <message name="depositResponse" />
  <portType name="AccountPortType">
    <operation name="withdraw">
      <input message="tns:withdraw" name="withdraw" />
      <output message="tns:withdrawResponse"
        name="withdrawResponse" />
    </operation>
    <operation name="deposit">
      <input message="tns:deposit" name="deposit" />
      <output message="tns:depositResponse"
        name="depositResponse" />
    </operation>
  </portType>
  ...
</definitions>
```

Client example

Example 45 shows a client that executes a transfer of funds as a transaction. After starting the transaction, the client withdraws \$1000 dollars from Bill's account and deposits the money into Ben's account.

Example 45: *Starting and Ending a Transaction on the Client Side*

```
// C++
...
IT_Bus::Bus_var bvar = IT_Bus::Bus::create_reference();
1 AccountClient acc;

try {
    // start a txn
2    bvar->begin("ots");
    acc.withdraw("Bill", 1000);
3    acc.deposit("Ben", 1000);
    bvar->commit(IT_TRUE,"ots");
    cout << "Transaction completed successfully." << endl;
}
4 catch(IT_Bus::Exception& e) {
    bvar->rollback("ots");
    cout << endl << "Caught Unexpected Exception: "
        << endl << e.Message() << endl;
    return -1;
}
```

The preceding transactional client code can be explained as follows:

1. The `AccountClient` object, `acc`, is a client proxy representing the `AccountPortType` port type.
2. The `IT_Bus::Bus::begin()` function initiates the transaction. The `ots` string, which is passed as the argument to `begin()`, specifies that the current transaction uses the CORBA OTS transaction factory.
3. The `IT_Bus::Bus::commit()` function attempts to commit the changes in the server (withdrawal and deposit of money).
4. If an exception is thrown, the transaction must be aborted by calling the `IT_Bus::Bus::rollback()` operation.

Message Attributes

This chapter describes how to program message attributes, which enable you to send extra data in a WSDL message during an operation call.

In this chapter

This chapter discusses the following topics:

Introduction to Message Attributes	page 108
Schemas	page 111
Name-Value API	page 113
Transport-Specific API	page 117
Using Message Attributes in a Client	page 120
Using Message Attributes in a Server	page 123

Introduction to Message Attributes

Overview

Message attributes provide a way of transmitting data in a WSDL message header as part of an operation invocation. For example, message attributes are useful in the context of secure communication, where they can be used to transmit authentication data between clients and servers.

Message attribute categories

Message attributes are properties that are set on an instance of a WSDL port. They are defined in a WSDL schema and are usually transport-specific. They can be divided into the following categories:

- Attributes that can be sent from the client to the server (*input message attributes*).
- Attributes that can be sent from the server to the client (*output message attributes*).

Additionally, the following kinds of message attribute can only be set locally and are not transmitted between applications:

- Attributes that configure the WSDL port on the client side (not transmitted).
- Attributes that configure the WSDL port on the server side (not transmitted).

Input and output messages

Figure 9 shows how message attributes are sent in the input message header, from client to server, and in the output message header, from server to client.

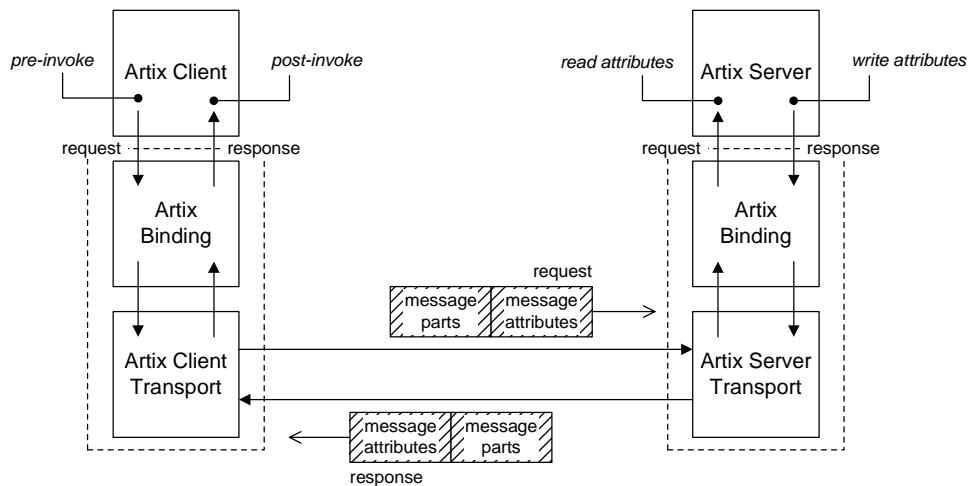


Figure 9: *Passing Message Attributes in Input and Output Messages*

Client interception points

A client can access message attributes at the following interception points:

- *Pre-invoke*—write input message attributes prior to an operation call.
- *Post-invoke*—read output message attributes after an operation call.

Server interception points

A server can access message attributes within the body of an operation implementation to do either of the following:

- Read the input message attributes received from the client.
- Write output message attributes to send to the client.

Oneway operations

A WSDL oneway operation defines only an input message. Hence, in a oneway operation it is only possible to define input message attributes.

Setting message attributes in configuration

It is possible to specify message attributes in configuration, by adding WSDL extension elements to the `<port>` element of the WSDL contract.

For example, the HelloWorld MQ Soap example (located in `ArtixInstallDir\artix\Version\demos\hello_world\mq_soap`) defines the `<port>` element in its WSDL contract as follows:

```
<definitions ... >
...
<service name="HelloWorldService">
  <port binding="tns:HelloWorldPortBinding"
        name="HelloWorldPort">
    <mq:client QueueManager="MY_DEF_QM"
              QueueName="HW_REQUEST"
              AccessMode="send"
              ReplyQueueManager="MY_DEF_QM"
              ReplyQueueName="HW_REPLY"
    />

    <mq:server QueueManager="MY_DEF_QM"
              QueueName="HW_REQUEST"
              ReplyQueueManager="MY_DEF_QM"
              ReplyQueueName="HW_REPLY"
              AccessMode="receive"
    />
  </port>
</service>
</definitions>
```

The attributes in the preceding example define the name and properties of an MQ series message queue both on the client side and the server side.

Setting message attributes by programming

Artix also allows you to set message attributes by programming. This gives you finer control over message attributes, enabling you to set them per-invocation instead of per-connection.

There are two styles of API for accessing and modifying message attributes by programming, as discussed in the following sections:

- ["Name-Value API" on page 113.](#)
- ["Transport-Specific API" on page 117.](#)

Schemas

Overview

The various kinds of message attributes are defined in a collection of XML schema definitions (one schema file for each transport type), located in the following directory:

ArtixInstallDir/artix/Version/schemas

Schema documentation

For documentation on the message attribute settings, see the relevant sections of *Designing Artix Solutions* concerning HTTP Transport Attributes, MQSeries Transport Attributes and Tibco Transport Attributes.

Schemas for message attributes

The message attributes supported by Artix are defined by transport-specific XSLT schema files, located in the *ArtixInstallDir/artix/Version/schemas* directory. The transport schemas with message attributes are listed in [Table 4](#).

Table 4: *Transport Schemas with Message Attributes*

Schema Type	File
HTTP	<i>ArtixInstallDir/artix/Version/schemas/http-conf.xsd</i>
MQ Series	<i>ArtixInstallDir/artix/Version/schemas/mq.xsd</i>
Tibco	<i>ArtixInstallDir/artix/Version/schemas/tibrv.xsd</i>

HTTP schema example

[Example 46](#) shows an extract from the HTTP schema, *http-conf.xsd*, showing some message attributes that can be set on the client side (that is, input message attributes).

The `UserName` and `Password` input message attributes can be used to send authentication data to a server. By default, these message attributes are sent in a BASIC HTTP authentication header.

Example 46: *Sample Extract from the http-conf.xsd Schema*

```
<xs:schema ... >
  <xs:complexType name="clientType">
    <xs:complexContent>
      <xs:extension base="wsdl:tExtensibilityElement">

        <xs:attribute name="UserName" type="xs:string"
          use="optional" />

        <xs:attribute name="Password" type="xs:string"
          use="optional" />

        ...
      </xs:extension>
      ...
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

Name-Value API

Overview

The name-value API is a transport-neutral API for setting and getting message attributes, where the attributes are stored in a table of name-value pairs. Attributes are identified by passing a string argument to one of the `set_Type()` or `get_Type()` functions (for a complete list of attribute identifiers, see the relevant schema in “Schemas for message attributes” on page 111).

This subsection discusses the following aspects of the name-value API:

- [Inheritance hierarchy](#).
- [MessageAttributes class](#).
- [NamedAttributes class](#).

Inheritance hierarchy

Figure 10 shows the inheritance hierarchy for the classes involved in the name-value API for message attributes.

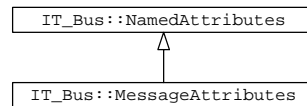


Figure 10: *Inheritance Hierarchy for IT_Bus::MessageAttributes Class*

MessageAttributes class

The `IT_Bus::MessageAttributes` class inherits functions for getting and setting name-value pairs from `IT_Bus::NamedAttributes`, but it does not define any new member functions of its own. The `MessageAttribute` class is used as the base class for transport-specific message attribute classes and instances of a `MessageAttribute` type encapsulate the settings for a specific transport.

NamedAttributes class

The `IT_Bus::NamedAttributes` class acts as a container for a collection of name-value pairs. The name in a name-value pair is a string identifier and the value is a data value whose type can be any of the basic WSDL data types.

The `IT_Bus::NamedAttribute` API, shown in [Example 47](#), provides a type-safe interface to the collection of name-value pairs using type-specific get and set operations, `get_Type()` and `set_Type()`.

Example 47: *The `IT_Bus::NamedAttribute` API*

```
// C++
IT_Bus::Boolean get_boolean(const IT_Bus::String& name) const
IT_THROW_DECL((WrongTypeException, NoSuchAttributeException));

void set_boolean(
    const IT_Bus::String& name,
    IT_Bus::Boolean data
);

IT_Bus::Byte get_byte(
    const IT_Bus::String& name
) const
IT_THROW_DECL((WrongTypeException, NoSuchAttributeException));

void set_byte(
    const IT_Bus::String& name,
    IT_Bus::Byte data
);

IT_Bus::Short get_short(
    const IT_Bus::String& name
) const
IT_THROW_DECL((WrongTypeException, NoSuchAttributeException));

void set_short(
    const IT_Bus::String& name,
    IT_Bus::Short data
);

IT_Bus::Int get_int(
    const IT_Bus::String& name
) const
IT_THROW_DECL((WrongTypeException, NoSuchAttributeException));

void set_int(
    const IT_Bus::String& name,
    IT_Bus::Int data
);

IT_Bus::Long get_long(
    const IT_Bus::String& name
```

Example 47: *The IT_Bus::NamedAttribute API*

```

) const
IT_THROW_DECL((WrongTypeException, NoSuchAttributeException));

void set_long(
    const IT_Bus::String& name,
    IT_Bus::Long data
);

IT_Bus::UByte get_ubyte(
    const IT_Bus::String& name
) const
IT_THROW_DECL((WrongTypeException, NoSuchAttributeException));

void set_ubyte(
    const IT_Bus::String& name,
    IT_Bus::UByte data
);

IT_Bus::UShort get_ushort(
    const IT_Bus::String& name
) const
IT_THROW_DECL((WrongTypeException, NoSuchAttributeException));

void set_ushort(
    const IT_Bus::String& name,
    IT_Bus::UShort data
);

IT_Bus::UInt get_uint(
    const IT_Bus::String& name
) const
IT_THROW_DECL((WrongTypeException, NoSuchAttributeException));

void set_uint(
    const IT_Bus::String& name,
    IT_Bus::UInt data
);

IT_Bus::ULong get_ulong(
    const IT_Bus::String& name
) const
IT_THROW_DECL((WrongTypeException, NoSuchAttributeException));

void set_ulong(
    const IT_Bus::String& name,

```

Example 47: *The IT_Bus::NamedAttribute API*

```

    IT_Bus::ULong data
    );

IT_Bus::Float get_float(
    const IT_Bus::String& name
) const
IT_THROW_DECL((WrongTypeException, NoSuchAttributeException));

void set_float(
    const IT_Bus::String& name,
    IT_Bus::Float data
);

IT_Bus::Double get_double(
    const IT_Bus::String& name
) const
IT_THROW_DECL((WrongTypeException, NoSuchAttributeException));

void set_double(
    const IT_Bus::String& name,
    IT_Bus::Double data
);

IT_Bus::String get_string(
    const IT_Bus::String& name
) const
IT_THROW_DECL((WrongTypeException, NoSuchAttributeException));

void set_string(
    const IT_Bus::String& name,
    const IT_Bus::String& data
);
...
const IT_Bus::NamedAttributes::StringList& get_names();

void clear_name_values();

```


Transport-Specific API

Overview

In addition to the neutral API for setting message attributes (as defined by `IT_Bus::NamedAttributes`), Artix also provides a transport-specific API for certain transports. This subsection describes the following aspects of transport-specific APIs:

- [Inheritance hierarchy](#).
- [Transports with a message attribute API](#).
- [Tibco transport example](#).

WARNING: If you decide to use a transport-specific API, you should note that your application will be tied to a specific transport; that is, you lose transport pluggability. You should consider carefully the impact that this might have on the design of your system before opting to use a transport-specific API.

Inheritance hierarchy

Figure 11 shows the inheritance hierarchy for the classes involved in the transport-specific API for message attributes.

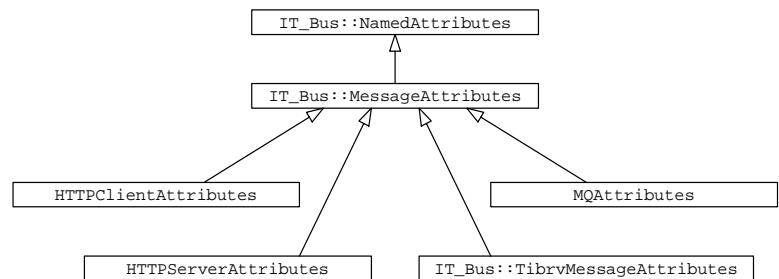


Figure 11: Inheritance Hierarchy for the Transport-Specific API

Transports with a message attribute API

The following transports provide a message attributes API:

- HTTP—there are two parts to this API, as follows:
 - ◆ Client side—defined by the `HTTPClientAttributes` class in the `<it_bus_config/http_wsdl_client.h>` header
 - ◆ Server side—defined by the `HTTPServerAttributes` class in the `<it_bus_config/http_wsdl_server.h>` header.
- MQ Series—defined by the `MQAttributes` class in the `<it_bus_config/mq_wsdl_port.h>` header.
- Tibco—defined by the `IT_Bus::TibrvMessageAttributes` class in the `<it_bus_config/tibrv_message_attributes.h>` header.

Tibco transport example

[Example 48](#), which is taken from the `<it_bus_config/tibrv_message_attributes.h>` header file, shows the transport-specific API for getting and setting message attributes on the Tibco transport.

Example 48: Getting and Setting Tibco Message Attributes

```
// C++
namespace IT_Bus
{
    class IT_BUS_API TibrvMessageAttributes
        : public virtual MessageAttributes
    {
    public:
        ...
        virtual const String& get_send_subject();
        virtual void set_send_subject(const String&
send_subject);

        virtual const String& get_reply_subject();
        virtual void set_reply_subject(
            const String& reply_subject
        );

        virtual const String& get_sender();
        virtual void set_sender(const String& sender);

        virtual const ULong& get_sequence();

        virtual const Double& get_time_limit();
    };
};
```

Example 48: *Getting and Setting Tibco Message Attributes*

```
virtual void set_time_limit(const Double& time_limit);  
  
virtual const UByte& get_jms_delivery_mode();  
  
virtual const UByte& get_jms_priority();  
  
virtual const ULong& get_jms_timestamp();  
  
virtual const ULong& get_jms_expiration();  
  
virtual const String& get_jms_type();  
  
virtual const String& get_jms_message_id();  
  
virtual const String& get_jms_correlation_id();  
  
virtual const Boolean& get_jms_redelivered();  
...  
};  
};
```

Using Message Attributes in a Client

Overview

This section describes how to write a client that sends message attributes across the wire to a server as part of an operation invocation.

How to use message attributes in a client

To use message attributes on the client side, perform the following steps:

Step	Action
1	Obtain an <code>IT_Bus::Port</code> object by calling <code>get_port()</code> on the client proxy object.
2	Call the <code>use_input_message_attributes()</code> and <code>use_output_message_attributes()</code> functions on the <code>IT_Bus::Port</code> object to initialize the message attribute functionality.
3	Pre-invoke step—set the input message attributes on the <code>IT_Bus::Port</code> object.
4	Invoke a WSDL operation on the client proxy.
5	Post-invoke step—read the output message attributes from the <code>IT_Bus::Port</code> object.

C++ example

To use message attributes in a sample client, you can modify the HelloWorld HTTP Soap client as shown in [Example 49](#). Edit the `client.cxx` file, which is located in the `ArtixInstallDir/artix/Version/demos/hello_world/http_soap/client` directory. In [Example 49](#), the client sets two input message attributes, `UserName` and `Password`, prior to the WSDL operation call and reads a single output message attribute, `ContentType`, after the call.

Example 49: Using Message Attributes in a Client

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

Example 49: *Using Message Attributes in a Client*

```

try
{
    IT_Bus::init(argc, argv);

    HelloWorldClient hw;

    String string_in;
    String string_out;

1    // Initialize message attributes.
    IT_Bus::Port& hw_port = hw.get_port();
    hw_port.use_input_message_attributes();
    hw_port.use_output_message_attributes();

2    // Pre-invoke: Set input message attributes.
    IT_Bus::MessageAttributes& hw_input =
        hw_port.get_input_message_attributes();
    hw_input.set_string("UserName", "nobody");
    hw_input.set_string("Password", "hushhush");

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

4    // Post-invoke: Read output message attributes.
    IT_Bus::MessageAttributes& hw_output =
        hw_port.get_output_message_attributes();
    try {
        String cont_type = hw_output.get_string("ContentType");
        cout << "Message attribute received: ContentType = " <<
cont_type << endl;
    }
5    catch (IT_Bus::NoSuchAttributeException) { }
}
catch(IT_Bus::Exception& e)
{
    cout << endl << "Caught Unexpected Exception: "
        << endl << e.Message()
        << endl;
    return -1;
}

```

The preceding client code example can be explained as follows:

1. The HelloWorld client proxy, `hw`, defines the `get_port()` method to give you access to the `IT_Bus::Port` object that controls the connection on the client side.

You switch on message attributes on the client side by calling

`use_input_message_attributes()` and

`use_output_message_attributes()` on the port object. By default, the message attribute feature is not enabled because it adds a certain performance penalty.

2. Pre-invoke interception point—the input message attribute object, `hw_input`, enables you to set attributes that are passed over the connection to the server.
3. The `sayHi()` operation performs the remote procedure call on the server.
4. Post-invoke interception point—the output message attribute object, `hw_output`, enables you to retrieve the attributes sent by the server.
5. The `IT_Bus::NoSuchAttributeException` exception is thrown if you try to read an output attribute that was not sent by the server.

Using Message Attributes in a Server

Overview

On the server side, message attributes can only be accessed within an *execution context*. That is, inside the body of a function that implements a WSDL operation.

This section describes how to write a server that receives input message attributes from a client and then sends output message attributes back to the client.

How to use message attributes in a server

To use message attributes on the server side, perform the following steps:

1. Override the port's virtual `activation()` method and call the port's `use_input_message_attributes()` and `use_output_message_attributes()` to initialize the message attribute functionality.
 2. Within an execution context, obtain an `IT_Bus::Current` object by calling `get_bus()->get_current()` on the server stub base object.
 3. Using the current object's `get_operation().get_port()` operation, obtain an `IT_Bus::Port` object.
 4. Within the server execution context, you can use the `IT_Bus::Port` object to do either of the following:
 - ◆ Read input message attributes.
 - ◆ Set output message attributes.
-

C++ example

To use message attributes in a server, you can modify the HelloWorld HTTP SOAP server as shown in [Example 50](#). Edit the `HelloWorldImpl.cpp` file, which is located in the `ArtixInstallDir/artix/Version/demos/hello_world/http_soap/server`

directory. In [Example 50](#), the client sets two input message attributes, `UserName` and `Password`, prior to the WSDL operation call and reads a single output message attribute, `ContentType`, after the call.

Example 50: *Using Message Attributes in a Server*

```

// C++
#include "HelloWorldImpl.h"
#include <it_cal/cal.h>
IT_USING_NAMESPACE_STD
using namespace IT_Bus;

1 void HelloWorldImpl::activate(IT_Bus::Port& port)
  {
    port.use_input_message_attributes(true);
    port.use_output_message_attributes(true);
  }

void HelloWorldImpl::sayHi(IT_Bus::String & Response)
  IT_THROW_DECL((IT_Bus::Exception))
  {
2    // Get a reference to the port.
    Current& current = get_bus()->get_current();
3    Port& port = current.get_operation().get_port();

4    // Read input message attributes.
    IT_Bus::MessageAttributes& hw_input =
      port().get_input_message_attributes();

5    try
      {
        IT_Bus::String user_name = hw_input.get_string("UserName");
        IT_Bus::String password = hw_input.get_string("Password");

        cout << "Message attributes received:" << endl;
        cout << "  username = " << user_name
          << ", password = " << password << endl;
      }
6    catch (IT_Bus::NoSuchAttributeException) { }

    cout << "HelloWorldImpl::sayHi called" << endl;

    Response = IT_Bus::String("Greetings from the Artix HelloWorld
      Server");
  }

```


Example 50: *Using Message Attributes in a Server*

7

```
// Set output message attributes.
IT_Bus::MessageAttributes& hw_output =
    port.get_output_message_attributes();
hw_output.set_string("ContentType", "text/xml");
}
```

The server code in [Example 50](#) can be explained as follows:

1. Override the port's `activate()` method to activate the use of message attributes. Regardless of the threading model your servant will be called back for each `Port` for each `activation()`.
2. The servant's `Current` object is obtained through the `Bus` object representing the server connection. The `get_bus()` operation is defined on the `IT_Bus::ServerStubBase` class, which is a base class of `HelloWorldImpl`. It returns a reference to the `Bus` object that represents the server connection.
3. The `get_port()` operation is defined on the `IT_Bus::Operation` class, which is accessed through the current object's `get_operation()` operation.

Note: You cannot call `get_port()` on the server stub if you are using the `MULTI_THREADED` threading model when the servant implementation is registered against multiple ports. The `get_port()` operation is currently supported for the following scenarios only:

- `MULTI_INSTANCE` threading model with multiple ports.
- `MULTI_THREADED` threading model with only a single port.

4. To read the input message attribute object on the server side, call `get_input_message_attributes()` on the server port object.
5. In this example, the server peeks at the value of the `UserName` and `Password` attributes. Normally, however, you would not bother to read the `UserName` and `Password` at this point because they would automatically be processed by the server's transport layer.
6. The `IT_Bus::NoSuchAttributeException` exception is thrown here if you try to read an input attribute that was not sent by the client.
7. You can send output message attributes back to the client by setting attributes on the output message attributes object, `hw_output`.

Dynamic Configuration

This section describes how you can dynamically modify a WSDL port's connection parameters by parsing and modifying the WSDL contract.

In this chapter

This chapter discusses the following topics:

Introduction to Dynamic Configuration	page 128
Dynamically Allocating IP Ports	page 130

Introduction to Dynamic Configuration

Overview

Dynamic configuration is an Artix mechanism that enables you to modify the port settings in a WSDL contract at runtime. This mechanism is facilitated by the `IT_WSDL` API, which is a C++ API for parsing WSDL. [Figure 12](#) shows an overview of the Artix dynamic configuration mechanism.

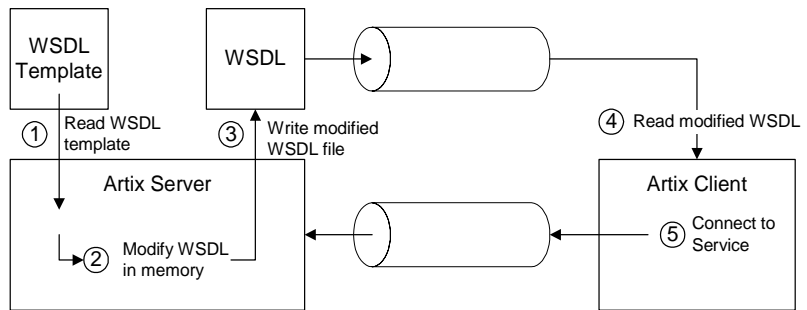


Figure 12: *Dynamic Configuration Mechanism*

Process of dynamic configuration

The process of dynamic configuration shown in [Figure 12](#) can be described as follows:

Stage	Description
1	As it starts up, the Artix server reads in a WSDL template file. The template is almost identical to the ultimate form of the WSDL contract, except that the <code><port></code> settings in the template are provisional only.
2	The server modifies the image of the WSDL template file in memory (represented as a WSDL parse tree). These modifications normally affect only the <code><port></code> settings.
3	The server writes out the modified WSDL to a new WSDL file, which is the form of the WSDL contract to be exposed to clients.

Stage	Description
4	When a client is about to use the service, it loads the modified WSDL file from the server side (typically through a HTTP URL).
5	The client connects to the service using the port settings it obtained from the modified WSDL contract.

Examples

This chapter describes the following examples of dynamic configuration:

- [“Dynamically Allocating IP Ports” on page 130.](#)

Dynamically Allocating IP Ports

Overview

This section describes how to program a server that uses dynamic IP port allocation. That is, when the connection parameters in a WSDL contract specify an IP port with the value 0. In this case, a client cannot read the IP port number from the original copy of the WSDL contract, because TCP/IP allocates a random IP port at runtime. The way to cope with this scenario is to program the server to write out a new copy of the WSDL contract which has the randomly-allocated IP port embedded in place of the 0 value.

Process for dynamically allocating IP ports

The process for dynamically allocating IP ports can be described as follows:

Stage	Description
1	When <code>IT_Bus::init()</code> is called on the server side, Artix activates all of the services that are currently registered.
2	During activation, Artix reads and parses the WSDL contracts for each of the registered services and ports. If a port address specifies an IP port value of 0, the TCP/IP transport randomly allocates an IP port on which it listens for connections. By default, Artix then modifies the WSDL parse tree in memory by replacing the 0 IP port value with the actual port number that was randomly assigned.
3	The server makes the randomly-assigned IP port value available to Artix clients by writing the modified WSDL parse tree to a file. You have to add some code to the server main function to perform this step.
4	When an Artix client starts up, it reads the modified WSDL file that is created in step 3, not the original WSDL file.

Modifying the HelloWorld demonstration

The example discussed here shows how you can modify the HelloWorld demonstration to perform dynamic IP port allocation. The source code to modify can be found in the following directory:

ArtixInstallDir/artix/Version/demos/HelloWorld/http_soap

How to implement dynamic IP port allocation

To implement dynamic IP port allocation, perform the following steps:

1. Modify the address in the WSDL contract to use IP port 0.

```
<definitions ... >
...
<service name="HelloWorldService">
  <port binding="tns:HelloWorldPortBinding"
    name="HelloWorldPort">
    <soap:address location="http://localhost:0"/>
  </port>
</service>
</definitions>
```

2. Add some code after `IT_Bus::init()` in the `server.cxx` file that writes the WSDL contract to a new file, `HelloWorld_written.wsdl`. For example, you could modify the main function of HelloWorld's `server.cxx` file as shown in [Example 51](#).

Example 51: Modified server.cxx File for Dynamic Port Allocation

```
// C++
...
int
main(int argc, char* argv[])
{
  cout << "HelloWorld Server" << endl;

  try
  {
    IT_Bus::init(argc, argv);

    IT_CurrentThread::sleep(2000);

    IT_Bus::Service * service = IT_Bus::Bus::get_service(
      QName("", "HelloWorldService", "http://xmlbus.com/HelloWorld")
    );
```

Example 51: *Modified server.cxx File for Dynamic Port Allocation*

```

const IT_WSDL::WSDLDefinitions & definitions =
    service->get_wsdl_definitions();

IT_Bus::FileOutputStream stream(
    "HelloWorld_written.wsdl"
);
IT_Bus::XMLOutputStream xml_stream(stream, true);

definitions.write(xml_stream);
stream.close();

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

return 0;
}

```

3. Modify the WSDL location in the client.

You must ensure that the client reads the WSDL file created in the previous step, `HelloWorld_written.wsdl`, which contains the actual value of the randomly-assigned IP port. In a typical deployment scenario, the client would read this file from the remote server host (for example, through a HTTP URL).

For the purpose of this simple demonstration, however, we assume that the client can read the WSDL contract, `HelloWorld_written.wsdl`, from a local directory. In this case, you

could modify the `client.cxx` file of the HelloWorld demonstration as follows:

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

    try
    {
        IT_Bus::init(argc, argv);
        HelloWorldClient hw(
            "HelloWorldServerDir/HelloWorld_written.wsdl"
        );
        ...
    }
}
```


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:

Simple Types	page 136
Complex Types	page 150
anyType Type	page 179
Nillable Types	page 184
SOAP Arrays	page 205
IT_Vector Template Class	page 217

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 137
String Type	page 138
QName Type	page 139
Date and Time Types	page 141
Decimal Type	page 142
Binary Types	page 144
Deriving Simple Types by Restriction	page 146
Unsupported Simple Types	page 149

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`. The Artix data types map closely to WSDL type names, and should be used by client applications.

Table of atomic types

The atomic types are:

Table 5: *Simple Schema Type to Simple Bus Type Mapping*

Schema Type	Bus Type
<code>xsd:boolean</code>	<code>IT_Bus::Boolean</code>
<code>xsd:byte</code>	<code>IT_Bus::Byte</code>
<code>xsd:unsignedByte</code>	<code>IT_Bus::UByte</code>
<code>xsd:short</code>	<code>IT_Bus::Short</code>
<code>xsd:unsignedShort</code>	<code>IT_Bus::UShort</code>
<code>xsd:int</code>	<code>IT_Bus::Int</code>
<code>xsd:unsignedInt</code>	<code>IT_Bus::UInt</code>
<code>xsd:long</code>	<code>IT_Bus::Long</code>
<code>xsd:unsignedLong</code>	<code>IT_Bus::ULong</code>
<code>xsd:float</code>	<code>IT_Bus::Float</code>
<code>xsd:double</code>	<code>IT_Bus::Double</code>
<code>xsd:string</code>	<code>IT_Bus::String</code>
<code>xsd:QName</code>	<code>IT_Bus::QName</code> <i>(SOAP only)</i>
<code>xsd:dateTime</code>	<code>IT_Bus::DateTime</code>
<code>xsd:decimal</code>	<code>IT_Bus::Decimal</code>
<code>xsd:base64Binary</code>	<code>IT_Bus::BinaryBuffer</code>
<code>xsd:hexBinary</code>	<code>IT_Bus::BinaryBuffer</code>

String Type

Overview

`xsd:string` maps to `IT_Bus::String`. `IT_Bus::String` is a typedef of the `IT_String` class (declared in `it_dsa/string.h`), which is an IONA implementation of the standard ANSI `String` class.

Codeset

Strings are assumed to be in the local codeset. If Artix writes a string as XML, however, it transcodes the string to the UTF-8 codeset.

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. The following member functions are useful when converting `IT_Bus::Strings` to ordinary C-style strings:

```
size_t length() const;
const char* c_str() const;
```

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();
```

Reference

For more details about C++ ANSI strings, see *The C++ Programming Language*, third edition, by Bjarne Stroustrup.

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)
hour offset from GMT	short	void setUTCTimeZoneOffset(short hour_offset, short minute_offset)
minute offset from GMT	short	void getUTCTimeZoneOffset(short & hour_offset, short & minute_offset)

The default constructor takes no parameters and initializes all of the fields to zero. 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)
```

Decimal Type

Overview

`xsd::decimal` maps to `IT_Bus::Decimal`, which is implemented by the IONA foundation class `IT_FixedPoint`, defined in `<it_dsa/decimal_point.h>`. `IT_FixedPoint` provides full fixed point decimal calculation logic using the standard C++ operators.

Note: Whereas `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 7](#) for a list of supported operators.

Table 7: *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;
}
```

Binary Types

Overview

There are two WSDL binary types, which map to C++ as shown in [Table 8:](#)

Table 8: *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 a similar set of member functions, as follows:

```
// C++
size_t get_length() const;

const IT_Bus::Byte get_data(const size_t pos) const;

void set_data(
    IT_Bus::Byte data[],
    size_t data_length,
    bool take_ownership = false
);
```

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 52](#) shows how a server might implement the `echoHexBinary` operation.

Example 52: *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))
{
    cout << "BaseImpl::echoHexBinary called" << endl;
    size_t length = inputHexBinary.get_length();
    Byte * the_data = new Byte[length];

    for (size_t idx = 0; idx < length; idx++)
    {
        the_data[idx] = inputHexBinary.get_data(idx);
    }

    Response.set_data(the_data, length, true);
}
```

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 53](#) 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 53: 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 54](#). The only values that can legally be set using the `set_value()` member function are the strings `RED`, `GREEN`, or `BLUE`.

Example 54: 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;
};
```

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:normalizedString
xsd:token
xsd:integer
xsd:positiveInteger
xsd:negativeInteger
xsd:nonNegativeInteger
xsd:nonPositiveInteger
xsd:time
xsd:duration
xsd:date
xsd:gMonth
xsd:gYear
xsd:gYearMonth
xsd:gDay
xsd:gMonthDay
xsd:anyURI
xsd:language
xsd:Name
xsd:NCName
xsd:QName (restricted support)
xsd:ENTITY
xsd:NOTATION
xsd:IDREF
```

Other Simple Types

```
xsd:list
xsd:union
```

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 151
Choice Complex Types	page 154
All Complex Types	page 158
Attributes	page 161
Nesting Complex Types	page 163
Deriving a Complex Type from a Simple Type	page 167
Occurrence Constraints	page 170
Arrays	page 174

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 [“Occurrence Constraints” on page 170](#).

WSDL example

[Example 55](#) shows an example of a sequence, `SequenceType`, with three elements.

Example 55: *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 55) to the `SequenceType` C++ class. An outline of this class is shown in Example 56.

Example 56: *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 57](#) shows how a client could use a proxy instance, `bc`, to invoke the `echoSequence` operation.

Example 57: 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 are currently not supported for choice complex types.

WSDL example

[Example 58](#) shows an example of a choice complex type, `ChoiceType`, with three elements.

Example 58: *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 58](#)) to the `SequenceType` C++ class. An outline of this class is shown in [Example 59](#).

Example 59: 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 59: *Mapping of ChoiceType to C++*

```

enum ChoiceTypeDiscriminator
{
    varFloat,
    varInt,
    varString,
    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 60](#) shows how a client could use a proxy instance, `bc`, to invoke the `echoChoice` operation.

Example 60: *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 60: *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:
            fail =(cIn.getvarFloat()!=cResult.getvarFloat());
            break;
        case ChoiceType::varInt:
            fail =(cIn.getvarInt()!=cResult.getvarInt());
            break;
        case ChoiceType::varString:
            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 61](#) shows an example of an all complex type, `AllType`, with three elements.

Example 61: *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 61](#)) to the `AllType` C++ class. An outline of this class is shown in [Example 62](#).

Example 62: 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 63](#) shows how a client could use a proxy instance, `bc`, to invoke the `echoAll` operation.

Example 63: *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 must conform to the following syntax:

```
<attribute name="AttrName" type="AttrType"/>
```

Limitations

The following attribute types are *not* supported:

- `xsd:IDREFS`
 - `xsd:ENTITY`
 - `xsd:ENTITIES`
 - `xsd:NOTATION`
 - `xsd:NMTOKEN`
 - `xsd:NMTOKENS`
-

WSDL example

[Example 64](#) shows how to define a sequence type with a single attribute, `prop`, of `xsd:string` type.

Example 64: Definition of a Sequence Type with an 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

[Example 65](#) shows an outline of the C++ `SequenceType` class generated from [Example 64 on page 161](#), which defines accessor and modifier functions for the `prop` attribute.

Example 65: Mapping an 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);
};
```

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 66](#).

WSDL example

[Example 66](#) shows an example of a nested complex type, which features a choice complex type, `NestedChoiceType`, nested inside a sequence complex type, `SeqOfChoiceType`.

Example 66: *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 67](#) shows an outline of the generated C++ `NestedChoiceType` class.

Example 67: 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 68](#) shows an outline of the generated C++ `SeqOfChoiceType` class, which shows how the nested complex type is mapped within a sequence complex type.

Example 68: Mapping of `SeqOfChoiceType` to C++

```
// C++
class SeqOfChoiceType : public IT_Bus::SequenceComplexType
{
    ...
```


Example 68: *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 63](#) shows how a client could use a proxy instance, `bc`, to invoke the `echoSeqOfChoice` operation.

Example 69: *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 69: *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 70](#) 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 70: *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 71](#) 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 71: *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 72](#) shows an outline of the C++ `internationalPrice` class generated from [Example 71 on page 168](#).

Example 72: *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 72: *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.

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 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 174](#).

Limitations

In the current version of Artix, occurrence constraints can be used only within the following complex types:

- all complex types,
- sequence complex types.

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. You should not use this type directly in your code. Use the `IT_Vector` (see [“IT_Vector Template Class” on page 217](#)) in place of `IT_Bus::ElementListT`. The `IT_Bus::ElementListT` types automatically convert to and from `IT_Vector` types.

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;

size_t get_max_occurs() const;

void set_size(size_t new_size);
```

```
size_t get_size() const;

const QName & get_item_name() const;
```

WSDL example

[Example 73](#) shows the definition of a sequence type, `SequenceType`, which contains a list of integer elements followed by a list of string elements.

Example 73: Sequence Type with 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 74](#) shows an outline of the C++ `SequenceType` class generated from [Example 73 on page 171](#), which defines accessor and modifier functions for the `varInt` and `varString` elements.

Example 74: 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;
```

Example 74: *Mapping of SequenceType to C++*

```

IT_Bus::ElementListT<IT_Bus::Int> & getvarInt();

void setvarInt(const IT_Bus::ElementListT<IT_Bus::Int> & val);

const IT_Bus::ElementListT<IT_Bus::String> & getvarString()
  const;

IT_Bus::ElementListT<IT_Bus::String> & getvarString();

void setvarString(const IT_Bus::ElementListT<IT_Bus::String> &
  val);

private:
  ...
};

```

IT_ElementListT is for internal use by the Artix generated code and should not be used directly in user developed code. Because the IT_Bus::ElementListT template supports automatic conversion to IT_Vector, you should treat the return values and arguments of the preceding integer and string accessor functions as if they were IT_Vector<IT_Bus::Int> and IT_Vector<IT_Bus::String> respectively.

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 `IT_Vector` 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_Vector<IT_Bus::Int>& v1 = seq.getvarInt();
IT_Vector<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).

Note: The `IT_Vector` class template does not provide the `set_size()` function. Hence, you cannot invoke `set_size()` on `v1` or `v2`.

References

For more details about vector types see:

- The “[IT_Vector Template Class](#)” on page 217.
- The section on C++ ANSI vectors in *The C++ Programming Language*, third edition, by Bjarne Stroustrup.

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 205](#).

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_Vector

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. The principal difference is that the C++ array class, *ArrayName*, can be treated as a vector.

For example, the C++ array class, *ArrayName*, supports the `size()` member function and individual elements can be accessed using the `[]` operator.

WSDL array example

[Example 75](#) shows how to define a one-dimensional string array, `ArrayOfString`, whose size can lie anywhere in the range 0 to unbounded.

Example 75: 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>
    </schema>
  </types>
  ...
</definitions>
```

C++ mapping

[Example 76](#) shows how the `ArrayOfString` string array (from [Example 75 on page 175](#)) maps to C++.

Example 76: Mapping of ArrayOfString to C++

```
// C++
class ArrayOfString : public IT_Bus::ArrayT<IT_Bus::String,
  &ArrayOfString_varString_qname, 0, -1>
{
public:
  ArrayOfString();
  ArrayOfString(size_t dimensions[]);
  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 76: *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 [“Occurrence Constraints” on page 170](#)). 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 77](#) 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 221](#)).

Example 77: *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 163](#) for a discussion of nested types). [Example 78](#) shows an example of how to define a two-dimensional string array, `ArrayOfArrayOfString`.

Example 78: *Definition of a Multi-Dimensional String Array*

```

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

```

Example 78: *Definition of a Multi-Dimensional String Array*

```

<schema ... >
  <complexType name="ArrayOfString">
    <sequence>
      <element name="varString" type="xsd:string"
        minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
  </complexType>
  <complexType name="ArrayOfArrayOfString">
    <sequence>
      <element name="nestArray"
        type="xsd: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 79](#) shows an example of how to allocate and initialize a multi-dimensional array, of `ArrayOfArrayOfString` type.

Example 79: *C++ Example for a Multi-Dimensional Array*

```

// C++
// Array of array of String
ArrayOfArrayOfString a2(2,2);

a2[0][0] = "ZeroZero";
a2[0][1] = "ZeroOne";
a2[1][0] = "OneZero";
a2[1][1] = "OneOne";

```

The `ArrayOfArrayOfString` class has a special constructor which allows you to specify the two array dimensions, as follows:

```
ArrayOfArrayOfString(size_t dimension0, size_t dimension1);
```

This constructor allocates the memory needed for an array of size `[dimension0][dimension1]`.

A more cumbersome alternative is to specify the array size as a list of dimensions, for example:

```
// C++
size_t extents[] = {2, 2};
ArrayOfArrayOfString a2(extents);
```

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 80](#) shows how an instance, `a2`, of `ArrayOfArrayOfString` type converts to an instance of `IT_Vector<ArrayOfString>` type by assignment.

Example 80: Converting a Multi-Dimensional Array to IT_Vector Type

```
// Array of array of String
ArrayOfArrayOfString a2(2,2);
...
// 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 217](#).
- The section on C++ ANSI vectors in *The C++ Programming Language*, third edition, by Bjarne Stroustrup.

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: Currently, the only binding that supports `xsd:anyType` is the CORBA binding.

Prerequisite for using anyType

A prerequisite for using the `xsd:anyType` is that your application must be built with the `WSDLFileName_wsdlTypesFactory.cxx` source file. This file is generated automatically by the WSDL-to-C++ compiler utility.

anyType syntax

To declare an `xsd:anyType` element, use the following syntax:

```
<element name="ElementName" [type="xsd:anyType"]>
```

The attribute setting, `type="xsd:anyType"`, is optional. If the `type` attribute is missing, the XML schema assumes that the element is of `xsd:anyType` by default.

C++ mapping

The WSDL-to-C++ compiler maps the `xsd:anyType` type to the `IT_Bus::AnyHolder` class in C++.

The `IT_Bus::AnyHolder` class provides member functions to insert and extract data values, as follows:

- [Inserting and extracting atomic types.](#)
- [Inserting and extracting user-defined types.](#)

Note: It is currently not possible to nest an `IT_Bus::AnyHolder` instance directly inside another `IT_Bus::AnyHolder` instance.

Inserting and extracting atomic types

To insert and extract atomic types to and from an `IT_Bus::AnyHolder`, use the member functions of the following form:

```
void set_AtomicTypeFunc(const AtomicTypeName&);
AtomicTypeName& get_AtomicTypeFunc();
const AtomicTypeName& get_AtomicTypeFunc();
```

For a complete list of the functions for the basic atomic types, see [“AnyHolder API” on page 182](#).

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 55 on page 151](#)), you can insert a `SequenceType` instance into an `IT_Bus::AnyHolder` as follows:

```
// C++
// Create an instance of SequenceType type.
SequenceType seq;
seq.setvarFloat(3.14);
seq.setvarInt(1234);
seq.setvarString("This is a sample SequenceType.");

// Insert the SequenceType value into an xsd:anyType.
IT_Bus::AnyHolder aH;
aH.set_any_type(seq);
```

To extract the `SequenceType` instance from the `IT_Bus::AnyHolder`, you need to perform a C++ dynamic cast:

```
// C++
...
// Extract the SequenceType value from the IT_Bus::AnyHolder.
IT_Bus::AnyType& base_extract = aH.get_any_type();

// Cast the extracted value to the appropriate type:
SequenceType& seq_extract
    = dynamic_cast<SequenceType&>(base_extract);
```

Accessing the type information

You can find out what type of data is contained in an `IT_Bus::AnyHolder` instance by calling the following member function:

```
const IT_Bus::QName & get_type() const;
```

Type information is set whenever an `IT_Bus::AnyHolder` instance is initialized. For example, if you initialize an `IT_Bus::AnyHolder` by calling `set_boolean()`, the type is set to be `xsd:boolean`. If you call `set_any_type()` with an argument of `SequenceType`, the type would be set to `xsd1:SequenceType`.

Note: Because the XML representation of `xsd:anyType` is not self-describing, some type information could be lost when an `anyType` is sent across the wire. In the case of a CORBA binding, however, there is no loss of type information, because CORBA `any`s are fully self-describing.

AnyHolder API

Example 81 shows the public API from the `IT_Bus::AnyHolder` class, including all of the function for inserting and extracting data values.

Example 81: 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 81: *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;
...
};
};
```

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 185
Nillable Atomic Types	page 187
Nillable User-Defined Types	page 191
Nested Atomic Type Nillable Elements	page 194
Nested User-Defined Nillable Elements	page 198
Nillable Elements of an Array	page 202

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 as a nillable element. If the `nillable` attribute is missing, the default is value is `false`.

On-the-wire format

On the wire, a nil value for an `<ElementName>` element is represented by the following XML fragment:

```
<ElementName xsi:nil="true"></ElementName>
```

Where the `xsi:` prefix represents the XML schema instance namespace, `http://www.w3.org/2001/XMLSchema-instance`.

C++ API for nillable types

[Example 82](#) shows the public member functions of the `IT_Bus::NillableValue` class, which provides the C++ API for nillable types.

Example 82: C++ API for Nillable Types

```
// C++
namespace IT_Bus
{
    template <class T, const QName* TYPE>
    class NillableValue : public Nillable
    {
    public:
        NillableValue();
        NillableValue(const NillableValue& other);
        explicit NillableValue(const T& other);
        virtual ~NillableValue();
        ...
        virtual const QName& get_type() const;
        virtual Boolean is_nil() const;
```

Example 82: *C++ API for Nillable Types*

```
...
virtual const T&
get() const IT_THROW_DECL((NoDataException));

virtual T&
get() IT_THROW_DECL((NoDataException));

virtual void set(const T& data);

virtual void reset();
...
};
...
};
```

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 185](#).

Table of nillable atomic types

[Table 9](#) shows how the XML schema atomic types map to C++ when the `xsd:nillable` flag is set to `true`.

Table 9: *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 9: *Nillable Atomic Types*

Schema Type	Nillable C++ Type
xsd:dateTime	IT_Bus::DateTimeNillable
xsd:decimal	IT_Bus::DecimalNillable
xsd:base64Binary	IT_Bus::BinaryBufferNillable
xsd:hexBinary	IT_Bus::BinaryBufferNillable

WSDL example

Example 83 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 83: *WSDL Example Showing Some Nillable Atomic Types*

```
<?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/">
      ...
      <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>
</definitions>
```


Example 83: WSDL Example Showing Some Nillable Atomic Types

```

</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 84 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 84: 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 185](#).

WSDL example

[Example 85](#) 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 85: 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 86 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 86: 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, &SOAPStructQName>
        SOAPStructNillable;
    typedef IT_Bus::NillablePtr<SOAPStruct, &SOAPStructQName>
        SOAPStructNillablePtr;
};
```

The API for the `SOAPStructNillable` type is defined in [“AnyHolder API” on page 182](#).

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++ modifier and accessor are then 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);
```

Note: Arrays with nillable elements are treated differently—see [“Nillable Elements of an Array”](#) on page 202.

WSDL example

[Example 87](#) defines a sequence complex type, `Nil_SOAPStruct`, which has some nillable elements, `varInt`, `varFloat`, and `varString`.

Example 87: 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/">
      ...
      <complexType name="Nil_SOAPStruct">
        <sequence>
          <element name="varInt" nillable="true"
            type="xsd:int"/>
```

Example 87: WSDL Example of a Sequence Type with Nilable Elements

```

        <element name="varFloat" nillable="true"
            type="xsd:float"/>
        <element name="varString" nillable="true"
            type="xsd:string"/>
    </sequence>
</complexType>
</schema>
</types>
...

```

C++ mapping

Example 88 shows how the `Nil_SOAPStruct` sequence complex type is mapped to C++. Note how the modifiers and accessors for the nilable member elements, `setElementName()` and `getElementName()`, take pointer arguments and return pointers instead of actual values. For example, the `getvarInt()` function returns a pointer to an `IT_Bus::Int` rather than an `IT_Bus::Int` value.

Example 88: 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);

    const IT_Bus::Float * getvarFloat() const;
    IT_Bus::Float * getvarFloat();
    void setvarFloat(const IT_Bus::Float * val);

    const IT_Bus::String * getvarString() const;
    IT_Bus::String * getvarString();
    void setvarString(const IT_Bus::String * val);

    virtual const IT_Bus::QName & get_type() const;
    ...
}
}

```

Example 88: C++ Mapping of the `Nil_SOAPStruct` Sequence Type

```

};

typedef IT_AutoPtr<Nil_SOAPStruct> Nil_SOAPStructPtr;

typedef IT_Bus::NillableValue<Nil_SOAPStruct,
&Nil_SOAPStructQName> Nil_SOAPStructNillable;

typedef IT_Bus::NillablePtr<Nil_SOAPStruct,
&Nil_SOAPStructQName> Nil_SOAPStructNillablePtr;
...
};

```

C++ example

The following C++ example shows how to create and initialize a `Nil_SOAPStruct` instance. Notice, for example, how the argument to `setvarInt()` is a pointer value, `&i`.

```

// 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++ modifier and accessor are then 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);
```

Note: Arrays with nillable elements are treated differently—see [“Nillable Elements of an Array” on page 202](#).

WSDL example

[Example 89](#) defines a sequence complex type, `Nil_NestedSOAPStruct`, which includes a nillable element of `SOAPStruct` type, `varSOAP`.

Example 89: WSDL Example of a Nillable All Type inside a Sequence Type

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BaseService"
  targetNamespace="http://soapinterop.org/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  ...
  xmlns:tns="http://soapinterop.org/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://soapinterop.org/xsd">
  <types>
    <schema targetNamespace="http://soapinterop.org/xsd"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <complexType name="SOAPStruct">
        <all>
          <element name="varFloat" type="xsd:float"/>
          <element name="varInt" type="xsd:int"/>
          <element name="varString" type="xsd:string"/>
        </all>
      </complexType>
    </schema>
  </types>
</definitions>
```

Example 89: WSDL Example of a Nillable All Type inside a Sequence Type

```

        </complexType>
        ...
        <complexType name="Nil_NestedSOAPStruct">
            <sequence>
                <element name="varInt" nillable="true"
                    type="xsd:int"/>
                <element name="varSOAP" nillable="true"
                    type="xsd:SOAPStruct"/>
            </sequence>
        </complexType>
        ...
    </schema>
</types>
    ...
    
```

C++ mapping

Example 90 shows how the `Nil_NestedSOAPStruct` sequence complex type is mapped to C++. Note how the `getvarSOAP()` function returns a pointer to a `SOAPStruct` rather a `SOAPStruct` value. Likewise, the `setvarSOAP()` function takes a `SOAPStruct` pointer as its argument.

Example 90: 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);

    const SOAPStruct * getvarSOAP() const;
    SOAPStruct *      getvarSOAP();
    void setvarSOAP(const SOAPStruct * val);

    virtual const IT_Bus::QName & get_type() const;
    ...
};
    
```

NillablePtr types

To help you manage the memory associated with nillable elements of user-defined type, *UserType*, the WSDL-to-C++ utility generates a nillable smart pointer type, *UserTypeNillablePtr*. The *NillablePtr* template types are similar to the `std::auto_ptr<>` template types from the Standard Template Library—see “Smart Pointers” on page 42.

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 91 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 91: The NillablePtr Template Class

```
// C++
namespace IT_Bus
{
    /**
     * Template implementation of Nillable as an auto_ptr.
     * T is the C++ type of data, TYPE is the data type QName.
     */
    template <class T, const QName* TYPE>
    class NillablePtr : public Nillable, public IT_AutoPtr<T>
    {
    public:
        NillablePtr();
        NillablePtr(const NillablePtr& other);
        NillablePtr(T* data);
        virtual ~NillablePtr();
        ...
        void set(const T* data);

        virtual Boolean is_nil() const;

        virtual const QName& get_type() const;
        ...
    };
```

Example 91: *The NillablePtr Template Class*

```
...
};
```

C++ example

The following C++ example shows how to create and initialize a `Nil_NestedSOAPStruct` instance. Notice, for example, how the argument passed to `setvarSOAP()` is 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;
}
```

Nilable 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 92](#) 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 92: 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 92: WSDL Example of an Array with Nilable 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 93](#) shows how the Nil_SOAPArray array complex type is mapped to C++. Note that the array elements are of IT_Bus::ShortNilable type.

Example 93: C++ Mapping of the Nil_SOAPArray Array Type

```

// C++
namespace INTEROP {
    class Nil_SOAPArray
        : public IT_Bus::ArrayT<IT_Bus::ShortNilable,
        &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::ShortNilable> &
        getitem() const;

        IT_Bus::ElementListT<IT_Bus::ShortNilable> &
        getitem();

        void
        setitem(const IT_Vector<IT_Bus::ShortNilable> & val);

        virtual const IT_Bus::QName &
        get_type() const;
    };
}

```

Example 93: C++ Mapping of the Nil_SOAPArray Array Type

```

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;
    }
}

```

SOAP Arrays

Overview

In addition to the basic array types described in [“Arrays” on page 174](#), 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 206
Multi-Dimensional Arrays	page 210
Sparse Arrays	page 213
Partially Transmitted Arrays	page 216

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 94 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 94: 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>
      ...
    </schema>
  </types>
</definitions>
```

Sample encoding

[Example 95](#) 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 95: 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 96](#) shows a C++ example of how to allocate and initialize an `ArrayOfSOAPString` instance with four elements.

Example 96: 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 210](#).
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 213](#)). 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 97](#) 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 97: 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 multi-dimensional SOAP array

[Example 98](#) 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 98: 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 99](#) shows a C++ example of how to allocate and initialize an `Array2OfInt` instance with dimensions, `[2,3]`.

Example 99: 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 99: *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 100](#) 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 100: *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 101](#) shows an example of how to initialize a sparse array of `Array2OfInt` type.

Example 101: *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 99 on page 211](#)). 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 102](#) shows an example of how to read a sparse array of `Array2OfInt` type.

Example 102: 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 103](#) shows the encoding of a partially transmitted `ArrayOfSOAPString` instance.

Example 103: *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 218
Summary of IT_Vector Operations	page 221

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 10](#) are *not* defined in `IT_Vector`.

Table 10: *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 104](#) shows how to use these basic vector operations to initialize an integer vector with the first one hundred integer squares.

Example 104: 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 104 on page 219](#) can be written in a more idiomatic style using vector iterators, as shown in [Example 105](#).

Example 105:*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 11](#) lists the member types defined in `IT_Vector<T>`.

Table 11: *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 12](#) lists the `IT_Vector` member functions returning iterators.

Table 12: *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 13](#) lists the `IT_Vector` element access operations.

Table 13: *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 14](#) lists the `IT_Vector` stack operations.

Table 14: *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 15](#) lists the `IT_Vector` list operations.

Table 15: *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 16](#) lists the other operations supported by `IT_Vector`.

Table 16: *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).

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 226
IDL Basic Type Mapping	page 228
IDL Complex Type Mapping	page 229
IDL Module and Interface Mapping	page 238

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 13](#).

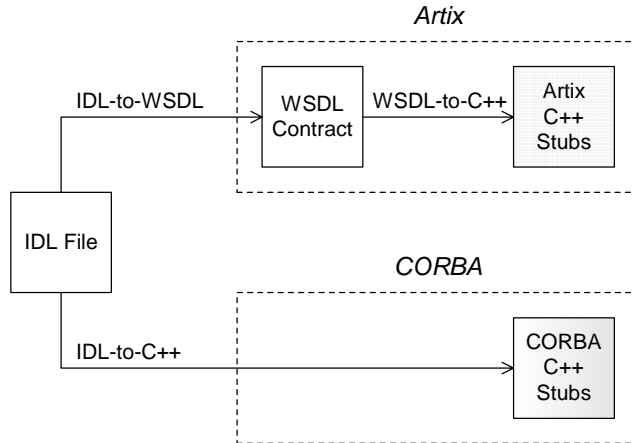


Figure 13: *Artix and CORBA Alternatives for IDL to C++ Mapping*

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 17 shows how IDL basic types are mapped to WSDL and then to C++.

Table 17: 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

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 146](#).

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 154](#).

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 151](#).

sequence types

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

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

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

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

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

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

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

Programming with Sequence Types

For details of how to use sequence types in C++, see [“Arrays” on page 174](#) and [“IT_Vector Template Class” on page 217](#).

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="xsd: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 174](#) and [“IT_Vector Template Class” on page 217](#).

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="xsd:SampleTypes.GenericExc"/>
...
<message name="_exception.SampleTypes.GenericExc">
  <part name="exception"
    element="xsd: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 [“Propagating Exceptions” on page 29](#).

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 61](#) for more details.

Operation mapping

[Example 106](#) 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 106: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 106 on page 240](#), map to C++ as shown in [Example 107](#),

Example 107: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 108](#) 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 108: Example IDL Attributes

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

The attributes from the preceding IDL, [Example 108 on page 241](#), map to C++ as shown in [Example 109](#),

Example 109: 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()`.

Index

Symbols

- <extension> tag 168
- <fault> tag 30
- <port> element 110
- <restriction> tag 167
- <simpleContent> tag 167

A

- abstract interface type 227
- all complex type
 - nillable example 191
- AllComplexType class 158
- all groups 158
- anonymous types
 - avoiding 163
- AnyHolder class 179
 - get_any_type() function 180
 - get_type() function 181
 - inserting and extracting atomic types 180
 - inserting and extracting user types 180
 - set_any_type() function 180
- AnyType class 180
- anyType type 179
 - nillable 187
- anyURI 149
- arrays
 - multi-dimensional native 176
 - native 174
 - SOAP 205
- arrayType attribute 207
- array types
 - nillable elements 202
- artix.cfg file 55
- Artix foundation classes 17
- Artix locator
 - overview 66
- Artix namespaces 5
- Artix services
 - locator 69
- ART library 17
- assign() 218
- at() 218
- atomic types 137

- nillable example 188
- nillable types 187
- attributes
 - mapping 161
- auto_ptr template 42

B

- Base64Binary type 144
- base64Binary type
 - nillable 188
- BASIC authentication 111
- begin() 103, 105
- begin_session() 91
- below_capacity() function 80
- binary types 144
 - get_data() 144
 - set_data() 144
- binding name
 - specifying to code generator 3
- boolean type
 - nillable 187
- bounded sequences 234
- boxed value type 227
- building Artix applications 179
- Bus library 17
- byte type
 - nillable 187

C

- C++ mapping
 - parameter order 24
 - parameters 23
- checked facets 146
- choice complex type 163
- ChoiceComplexType class 154
- choice complex types 154
- clear() 218
- client
 - developing 12
 - proxy object 12
 - stub code, files 2
- client proxies

- and multi-threading 53
 - and threading 52
 - get_port() 120
 - client stub code 2
 - Code generation 2
 - code generation
 - from the command line 3
 - impl flag 8
 - code generator
 - command-line 3
 - files generated 2
 - codeset 138
 - commit() 103, 105
 - compare() 142
 - compiler requirements 17
 - complex datatypes
 - generated files 2
 - complex type
 - deallocating 41
 - deriving from simple 167
 - complex types 150
 - assignment operators 39
 - copying 39
 - nesting 163
 - recursive copying 40
 - configuration
 - message attributes 110
 - ORBname switch 75
 - ConnectException type 28
 - ContentType message attribute 124
 - CORBA
 - abstract interface 227
 - any 228
 - basic types 228
 - boolean 228
 - boxed value 227
 - char 228
 - enum type 229
 - exception type 235
 - fixed 228
 - forward-declared interfaces 227
 - local interface type 227
 - Object 228
 - sequence type 233
 - string 228
 - struct type 232
 - typedef 236
 - union type 230, 234
 - value type 227
 - wchar 228
 - wstring 228
 - CORBA binding 179
 - CosTransactions::Coordinator class 103
 - create_server() 49
 - create_service() 59
 - c_str() 138
- D**
- date 149
 - dateTime type
 - nillable 188
 - decimal type
 - nillable 188
 - declaration specifiers 19
 - declspec option 19
 - derivation
 - by extension 167
 - by restriction 167
 - get_simpleTypeValue() 169
 - set_simpleTypeValue() 169
 - DeserializationException type 28
 - destroy_server() 50
 - developing a server 8
 - DLL
 - building stub libraries 19
 - double type
 - nillable 187
 - duration 149
 - dynamic configuration
 - implementing 131
 - introduction to 128
 - of IP ports 130
- E**
- element 128
 - ElementListT class 170
 - conversion to IT_Vector 172
 - embedded mode
 - compiling 17
 - linking 17
 - encoding of SOAP array 211
 - EndpointNotExist fault 71
 - endpoint reference 62
 - endpoints 68
 - below_capacity() function 80
 - pausing and resuming 80
 - reached_capacity() function 80

- registering with the locator 75
- end_session() 97
- ENTITIES type 161
- ENTITY 149
- ENTITY type 161
- enumeration facet 146
- enum type 229
- Error() function 27
- exception
 - propagating 29
 - raising a fault exception 30
- exception handling
 - CORBA mapping 235
- Exception type 27
- exception type 235

F

- facets 146
 - checked 146
- FaultException type 29
- fixed decimal
 - compare() 142
 - DigitIterator 143
 - is_negative() 142
 - left_most_digit() 142
 - number_of_digits() 142
 - past_right_most_digit() 142
 - round() 142
 - scale() 142
 - truncate() 142
- float type
 - nillable 187
- forward-declared interfaces 227
- fractionDigits facet 146

G

- gDay 149
- get_all_endpoints() 92
- get_any_type() function 180
- get_bus() 125
- get_data() 144
- get_discriminator() 231
- get_discriminator_as_uint() 231
- getendpoints() 93
- get_extents() 207, 212, 215
- get_input_message_attributes() 95, 125
- get_item_name() 171
- get_max_occurs() 170

- get_min_occurs() 170
- get_port() 94, 120
- getsession_id() 91
- get_simpleTypeValue() 169
- get_size() 171
- get_threading_model() 50, 58
- get_type() function 181
- get_wsdl_location() 49
- gMonth 149
- gMonthDay 149
- gYear 149
- gYearMonth 149

H

- HelloWorld port type 6
- HexBinary type 144
- hexBinary type
 - nillable 188
- high water mark 55
- high_water_mark configuration variable 56
- HTTP
 - BASIC authentication 111
 - example port 13
- HTTPClientAttributes class 118
- http-conf.xsd file 111
- http plug-in 75
- HTTPServerAttributes class 118

I

- IDL
 - bounded sequences 234
 - enum type 229
 - exception type 235
 - object references 239
 - oneway operations 241
 - sequence type 233
 - struct type 232
 - typedef 236
 - union type 230, 234
- IDL attributes
 - mapping to C++ 241
- IDL basic types 228
- IDL interfaces
 - mapping to C++ 238
- IDL modules
 - mapping to C++ 238
- IDL operations
 - mapping to C++ 240

- parameter order 241
- return value 241
- IDL readonly attribute 242
- IDL-to-C++ mapping
 - Artix and CORBA 226
- IDL types
 - unsupported 227
- idl utility 226
- IDREF 149
- IDREFS type 161
- init() 130
 - ORBname parameter 79
- init() function 10, 12
- Initializing the Bus 10
- initial_threads configuration variable 56
- inout parameter ordering 25
- inout parameters 241
- in parameters 241
- input message 22
- input message attributes 108
- input parameters 22
- instance namespace 185
- integer 149
- interception points 109
- int type
 - nillable 187
- InvalidRouteException type 28
- IOException type 28
- IONA foundation classes 17
- IP port
 - 0 value 130
 - implementing dynamic allocation 131
- IP ports
 - dynamically allocating 130
- is_empty() 215
- is_negative() 142
- is_nil() function 190, 193, 200
- IT_AutoPtr template 42
- IT_Bus::AllComplexType 158
- IT_Bus::Base64Binary 144
- IT_Bus::BinaryBuffer 137
- IT_Bus::Boolean 137
- IT_Bus::Bus::register_server_factory() 49
- IT_Bus::Byte 137
- IT_Bus::ChoiceComplexType 154
- IT_Bus::ConnectException 28
- IT_Bus::DateTime 137, 141
- IT_Bus::Decimal 137, 142
- IT_Bus::Decimal::DigitIterator 143
- IT_Bus::DeserializationException 28
- IT_Bus::Double 137
- IT_Bus::ElementListT 170
 - conversion to IT_Vector 172
- IT_Bus::Exception 27
- IT_Bus::Exception::Error() 27
- IT_Bus::Exception::Message() 27
- IT_Bus::Exception type 27
- IT_Bus::FaultException 29
- IT_Bus::Float 137
- IT_Bus::HexBinary 137, 144
- IT_Bus::init() 10, 12
 - activating services 130
- IT_Bus::Int 137
- IT_Bus::IOException 28
- IT_Bus::Long 137
- IT_Bus::MessageAttributes class 113
- IT_Bus::NamedAttributes class 113
- IT_Bus::NoSuchAttributeException exception 122, 125
- IT_Bus::QName 137
- IT_Bus::run() 11, 12
- IT_Bus::SequenceComplexType 151
- IT_Bus::SerializationException 28
- IT_Bus::ServiceException 28
- IT_Bus::Short 137
- IT_Bus::shutdown() 14
- IT_Bus::SoapEncArrayT 207
- IT_Bus::String 137, 138
- IT_Bus::String::iterator 138
- IT_Bus::TibrvMessageAttributes class 118
- IT_Bus::TransportException 28
- IT_Bus::UByte 137
- IT_Bus::UInt 137
- IT_Bus::ULong 137
- IT_Bus::UShort 137
- IT_BUS_E_FAULT error code 27
- IT_Bus namespace 5
- IT_Bus_Services::renewSessionFaultException 96
- IT_Bus_Services::SessionID 91
- iterators
 - in IT_Vector 219
- IT_FixedPoint class 142
- IT_HTTP_E_ACCESS_DENIED error code 27
- IT_HTTP_E_BAD_CONFIG error code 27
- IT_HTTP_E_COMM_ERROR error code 27
- IT_HTTP_E_NOT_FOUND error code 27
- IT_HTTP_E_SHUTTING_DOWN error code 27
- IT_Routing::InvalidRouteException 28

- IT_String class 138
 - IT_Vectorof class
 - resize() 218
 - IT_Vector class 170, 172
 - and set_size() 173
 - assign() 218
 - at() 218
 - clear() 218
 - converting to 178
 - differences from std::vector 218
 - iterators 219
 - operations 221
 - overview 217
 - resize() 218
 - IT_WSDL namespace 5
- L**
- language 149
 - leaks
 - avoiding 42
 - left_most_digit() 142
 - length() 138
 - length facet 146
 - libraries
 - Artix foundation classes 17
 - ART library 17
 - Bus 17
 - IONA foundation classes 17
 - license
 - display current 4
 - linker requirements 17
 - list 149
 - load balancing
 - with the locator 67
 - local interface type 227
 - locator
 - binding and protocol 69
 - demonstration code 67
 - embedded deployment 68
 - EndpointNotExist fault 71
 - load balancing 67, 68
 - LocatorService port type, C++ mapping 72
 - lookupEndpointResponse type 71
 - lookupEndpointResponse type, C++ mapping 74
 - lookupEndpoint type 71
 - lookupEndpoint type, C++ mapping 73
 - reading a reference from 76
 - registering endpoints 75
 - standalone deployment 68
 - WSDL contract 69
 - locator, Artix 66
 - locator_endpoint plug-in 75, 80
 - LocatorService port type 72
 - long type
 - nillable 187
 - lookupEndpointResponse type 71
 - lookupEndpointResponse type, C++ mapping 74
 - lookupEndpoint type 71
 - lookupEndpoint type, C++ mapping 73
 - low water mark 55
 - low_water_mark configuration variable 56
- M**
- mapping
 - IDL attributes 241
 - IDL interfaces 238
 - IDL modules 238
 - IDL operations 240
 - IDL to C++ 226
 - maxExclusive facet 146
 - maxInclusive facet 146
 - maxLength facet 146
 - maxOccurs 170, 174
 - max_size() 218
 - memory management 33
 - client side 35
 - copying and assignment 39
 - deallocating 41
 - rules 34
 - server side 36
 - smart pointers 42
 - Message() function 27
 - message attributes
 - categories 108
 - client example 120
 - ContentType 124
 - HTTPClientAttributes class 118
 - HTTPServerAttributes class 118
 - in configuration 110
 - input message 108
 - interception points 109
 - IT_Bus::TibrvMessageAttributes class 118
 - MQAttributes class 118
 - MQ series 110
 - name-value API 113
 - NoSuchAttributeException exception 122
 - oneway operation 109
 - output 108

- schemas 111
- server example 123
- transport-specific API 117
- MessageAttributes class 113
- messages
 - input 22
 - output 22
- minExclusive facet 146
- minInclusive facet 146
- minLength facet 146
- minOccurs 170
- mq.xsd file 111
- MQAttributes class 118
- MQ series
 - message attributes 110
- multi-dimensional native arrays 176
- MULTI_INSTANCE threading model 46, 54, 125
- multiple ports
 - per service 46
- multiple servants per port 46
- multiple services 46
- MULTI_THREADED threading model 55, 125
- multi-threading
 - client side 52
 - server side 54

N

- Name 149
- NamedAttributes class 113
- namespace
 - for generated C++ code 3
- namespaces
 - IT_Bus 5
 - IT_WSDL 5
 - using in C++ 5
- name-value API 113
- native arrays 174
- NCName 149
- negativeInteger 149
- nesting complex types 163
- nillable atomic member elements 194
- NillablePtr template class 200
- nillable types 194
 - atomic type, example 188
 - atomic types 187
 - IT_Bus::NillableValue 185
 - nillable array elements 202
 - NillablePtr template class 200
 - nillable user-defined member elements 198

- overview 184
- syntax 185
- user-defined types 191
 - xsi:nil attribute 185
- NillableValue class 185
- NMTOKENS type 161
- NMTOKEN type 161
- nonNegativeInteger 149
- nonPositiveInteger 149
- normalizedString 149
- NoSuchAttributeException exception 122, 125
- NOTATION 149
- NOTATION type 161
- number_of_digits() 142

O

- object references
 - mapping to C++ 239
- occurrence constraints
 - get_item_name() 171
 - get_max_occurs() 170
 - get_min_occurs() 170
 - get_size() 171
 - in all groups 158
 - in choice groups 154
 - in sequence groups 151
 - overview of 170
 - set_size() 170
- offset attribute 216
- oneway operations
 - in IDL 241
- operations
 - declaring 22
- ORBname, parameter to IT_Bus::init() 79
- ORBname command-line parameter 75
- ORBname command-line switch 55
- order of parameters 24
- OTS
 - transaction support 100
- out parameters 241
- output message 22
- output message attributes 108
- output parameters 22

P

- parameters
 - in IDL-to-C++ mapping 241
- parse tree

- WSDL 130
- partially transmitted arrays 216
- Password attribute 111
- past_right_most_digit() 142
- pattern facet 146
- plug-ins
 - http 75
 - locator_endpoint 75
 - locator_endpoint plug-in 80
 - soap 75
- plugins:sm_simple_policy:max_session_timeout 91
- plugins:sm_simple_policy:min_session_timeout 91
- port
 - specifying on the client side 12
 - specifying to code generator 3
- port object
 - use_input_message_attributes() 120, 123
 - use_output_message_attributes() 123
- ports
 - and endpoints 68
- port type
 - specifying to code generator 3
- positiveInteger 149
- propagating exceptions 29
- properties
 - in a reference 65
- proxies
 - constructor for references 79
- proxy object
 - and multi-threading 53
 - constructors 12

Q

- QName 149
- QName type
 - nillable 187

R

- reached_capacity() function 80
- recursive copying 40
- recursive deallocating 41
- ref:Reference type 71
- reference
 - to an endpoint 62
- references
 - constructor for client proxies 79
 - CORBA mapping 239
 - IT_Bus

Reference class 65

- looking up in the locator 68
- properties 65
- reading from the locator 76
- ref:Reference type 71
 - schema 71
 - static 63
 - transient 64
 - XML schema 62
- register_server factory() 49
- renew_session() 96
- resize() 218
- resources
 - server side 100
- rollback() 103, 105
- rollback_only() 103
- round() 142
- run() function 11, 12
- Running the Bus 11

S

- scale() 142
- schema
 - for references 71
- schemas 111
 - for references 62
- sequence complex type 163
- SequenceComplexType class 151
- sequence complex types 151
 - and arrays 174
- sequence type 233
- Serialization type 28
- servant
 - and threading models 54
- servants
 - multiple per port 46
- server
 - developing 8
 - implementation class 8
 - main() function 10
 - skeleton code, files 2
- server factory
 - creating 49
 - default implementation 46
 - deregistering services 49
 - implementing 46
 - multiple ports 46

- multiple services 46
 - registering a service 49
 - ServerFactoryBase class 58
 - ServerFactoryBase class 58
 - server skeleton code 2
 - service
 - registering in a server factory 49
 - specifying on the client side 12
 - ServiceException type 28
 - service name
 - specifying to code generator 3
 - SessionManagerClient 90
 - set_any_type() function 180
 - set_data() 144
 - setendpoint_group() 91
 - setpreferred_renew_timeout() 91
 - setsession_id() 92
 - set_simpleTypeValue() 169
 - set_size() 170, 173
 - set_timeout() 103
 - short type
 - nillable 187
 - shutdown() function 14
 - Shutting the Bus down 11
 - simple types
 - deriving by restriction 146
 - skeleton code
 - files 2
 - generating with wsdltocpp 3
 - smart pointer
 - assignment semantics 43
 - smart pointers 42
 - SOAP arrays 205
 - encoding 211
 - get_extents() 207, 212
 - multi-dimensional 210
 - one-dimensional 207
 - partially transmitted 216
 - sparse 213
 - syntax 206
 - SOAP bindings 69
 - SOAP-ENC:Array type 206
 - SOAP-ENC:offset attribute 216
 - SoapEncArrayT class 207
 - soap plug-in 75
 - sparse arrays 213
 - get_extents() 215
 - initializing 214
 - is_empty() 215
 - static reference 63
 - std::vector class 217
 - strings
 - codeset 138
 - c_str() 138
 - iterator 138
 - IT_String class 138
 - length() 138
 - string type
 - nillable 187
 - Stroustrup, Bjarne 138
 - struct type 232
 - stub code
 - files 2
 - stub libraries
 - building on Windows 19
- ## T
- threading
 - client proxy in two threads 52
 - get_threading_model() function 50
 - MULTI_INSTANCE model 54, 125
 - MULTI_THREADED model 55, 125
 - work queue 54
 - threading model
 - changing 58
 - create_service() 59
 - default 55
 - thread pool
 - configuration settings 55
 - initial threads 55
 - thread_pool:high_water_mark configuration variable 56
 - thread_pool:initial_threads configuration variable 56
 - thread_pool:low_water_mark configuration variable 56
 - Tibco transport 118
 - tibrv.xsd file 111
 - time 149
 - token 149
 - totalDigits facet 146
 - transaction factory 100
 - transaction factory name 102
 - transactions
 - begin() 103, 105
 - client example 104
 - commit() 103, 105
 - compatibility with CORBA OTS 101
 - CosTransactions::Coordinator class 103

- in Artix 100
- IT_Bus::Bus class 102
- OTS-based 100
- rollback() 103, 105
- rollback_only() 103
- set_timeout() 103
- transaction factory 100
- within_transaction() 103

transient references 64

TransportException type 28

transports

- schemas 111
- Tibco 118

truncate() 142

Tuxedo

- example port 13

typedef 236

U

union 149

union type 230, 234

unsignedByte type

- nillable 187

unsignedInt type

- nillable 187

unsignedLong type

- nillable 187

unsignedShort type

- nillable 187

unsupported IDL types 227

URL

- for WSDL contract 129
- for WSDL file 132

use_input_message_attributes 94

use_input_message_attributes() 120, 122, 123

use_output_message_attributes() 122, 123

user defined exceptions

- propagation 29

user-defined types

- nillable 191

UserName attribute 111

V

value type 227

_var types 43

W

wchar type 227

whiteSpace facet 146

within_transaction() 103

work queue 54

WSDL

- anyType syntax 179
- atomic types 137
- attributes 161
- binary types 144
- complex types 150
- deriving by restriction 146
- parse tree 130

wsdl:arrayType attribute 207

WSDL contract

- location of 13
- see WSDL file

WSDL facets 146

WSDL faults 235

WSDL file

- location 46, 49
- template for 128

wsdltocpp

- command-line options 3
- command-line switches 3
- files generated 2

wsdltocpp utility 179, 226

- declspec option 19
- generating default server factory 46

wstring type 227

X**xsd**

- anyURI 149
- date 149
- duration 149
- ENTITY 149
- gDay 149
- gMonth 149
- gMonthDay 149
- gYear 149
- gYearMonth 149
- IDREF 149
- language 149
- list 149
- Name 149
- NCName 149
- negativeInteger 149
- nonNegativeInteger 149
- nonPositiveInteger 149
- normalizedString 149

INDEX

- NOTATION 149
- positiveInteger 149
- QName 149
- time 149
- token 149
- union 149
- xsd:boolean 147
- xsd:dateTime type 141
- xsd:decimal type 142
- xsd:ENTITIES 161
- xsd:ENTITY 161
- xsd:IDREFS 161
- xsd:NMTOKEN 161
- xsd:NMTOKENS 161
- xsd:NOTATION 161
- xsd
 - integer 149
- xsi:nil attribute 185
- xsi namespace 185

