C++ Programmer's Guide

Version 1.1, July 2003

Making Software Work Together™

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: 20-Aug-2003

M 3 1 0 5

Contents

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*
- *Artix User's Guide*
- *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. *Artix User's Guide*

   The user's guide describes the development pattern for designing and deploying Artix enabled systems. It provides detailed examples for a number of typical use cases.

3. *Artix Tutorial Guide*

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

4. 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 doc-feedback@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.

CHAPTER 1

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

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.

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

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

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

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 -w artix_contract [-e web_service_name] [-t port] [-b
   binding_name] [-d output_dir] [-n namespace] [-impl] [-v]
   [-license] [-?]
```

You must specify the `-w` flag and the location of a valid Artix contract for the code generator to work. You can also supply the following optional parameters:

`-e web_service_name`	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.
`-t port`	Specifies the name of the port for which code is generated. The default is to used the first port listed in the contract.
`-b binding_name`	Specifies the name of the binding to use when generating code. The default is the first binding listed in the contract.
`-d output_dir`	Specifies the directory to which the generated code is written. The default is the current working directory.
`-n namespace`	Specifies the C++ namespace to use for the generated code.
`-impl`	Generates the skeleton code for implementing the server defined by the contract.
`-v`	Displays the version of the tool.
`-license`	Displays the currently available licenses.
`-?`	Displays help on using the command line tool.

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 107.
- 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, `wsdltocpp`, will generate an implementation class for your server when passed the `-impl` command flag.

Generated code

The implementation class code consists of two files:

*PortTypeName***Impl.h** contains the signatures and data types needed for the server implementation.

*PortTypeName***Impl.cxx** contains empty shells for the methods that implement the operations defined in the contract, as well as an empty contstructor 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 *`PortTypeName`*`Impl.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 10 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 *PortTypeName*`Client.h` and *PortTypeName*`Client.cxx`. You should never make any modifications to the generated code in these files. You also need to reference the files *PortTypeName*`.h` and *PortTypeName*`Types.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, *PortTypeName*`Client`, 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 11.

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:

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

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 12, 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 *PortName*`Client.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 `IT_Bus::Float`, etc. Required for all applications that use Artix functionality.

`it_ifc.lib`	`libit_ifc.so`
	`libit_ifc.sl`
	`libit_ifc.a`
		The 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.

CHAPTER 2

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

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 20).
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 20).
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 20) 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>
    ...
</definitions>
```

C++ mapping of testReverseParams

Example 10 shows how the preceding WSDL `testReverseParams` operation (from Example 9 on page 22) 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

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
`IT_HTTP_E_COMM_ERROR`	A communication error occurred.
`IT_HTTP_E_ACCESS_DENIED`	Username or password validation error by the server.
`IT_HTTP_E_BAD_CONFIG`	The configuration file is not valid.
`IT_HTTP_E_NOT_FOUND`	The URL or file was not found.
`IT_HTTP_E_SHUTTING_DOWN`	The system is entering a quiescent state.
`IT_BUS_E_FAULT`	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*

```
<?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/">
1           <complexType name="SampleFaultData">
                <all>
                    <element name="lowerBound" type="xsd:int"/>
                    <element name="upperBound" type="xsd:int"/>
                </all>
            </complexType>
            ...
        </schema>
    </types>
2   <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*

```
            <output message="tns:echoIntegerResponse"
                    name="echoIntegerResponse"/>
3           <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 *ExceptionMessage*`Exception`.

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

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

Client example

Example 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 37 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 32 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 40.

Recursive deallocation

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

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

Smart Pointers

Overview

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

- What is a smart pointer?
- Artix smart pointers.
- 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 *PortType*`Impl.h` and *PortType*`Impl.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 *PortType*`Impl.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 45.

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

```
// C++
...
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",
       "http://schemas.iona.com/idl/HelloWorld.idl");
3       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",
       "http://schemas.iona.com/idl/HelloWorld.idl");
5       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()
    {
```

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_ALTHelloWorldPort` 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 52.
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 52.
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

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 56). 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.

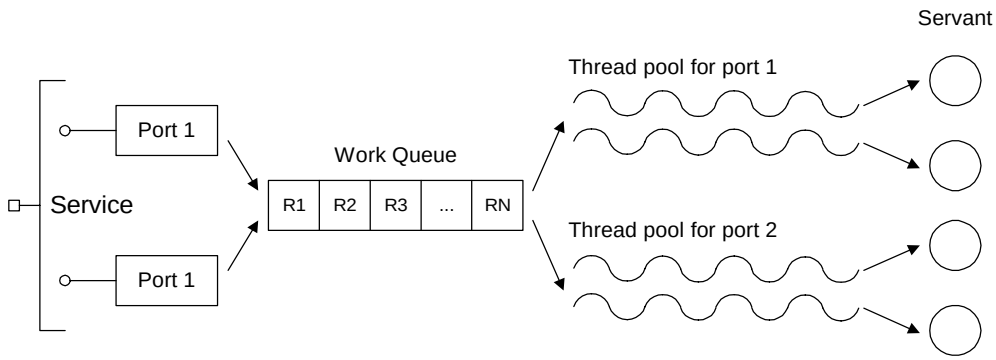

Figure 1: *Outline of the MULTI_INSTANCE Threading Model*

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.

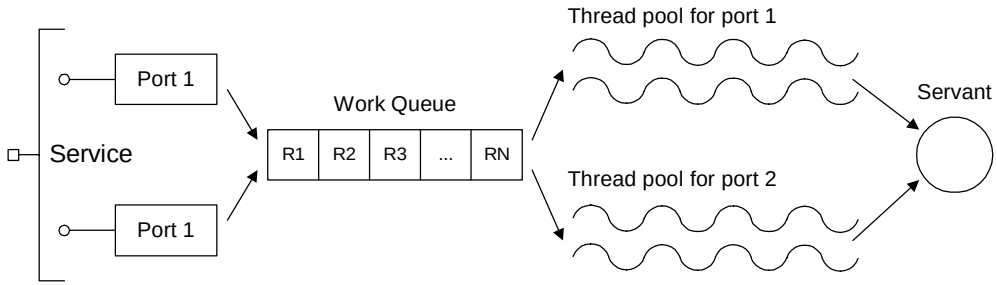

Figure 2: *Outline of the MULTI_THREADED Threading Model*

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();
};
```

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`	`create_server()` is called once for each thread in the thread pool (see "Thread pool configuration levels" on page 54).
`MULTI_THREADED`	`create_server()` 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();
};
```

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()
{
    return IT_Bus::MULTI_THREADED;
}
```

CHAPTER 3

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

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 3.

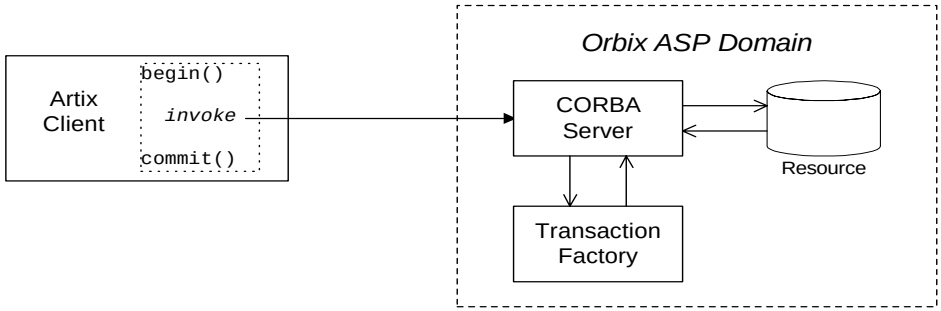

Figure 3: *Artix Client Invokes a Transactional Operation on a CORBA OTS Server*

In Figure 3, 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 28 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 28: *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 29 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 29: *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);
    acc.deposit("Ben", 1000);
3   bvar->commit(IT_TRUE,"ots");
    cout << "Transaction completed successfully." << endl;
}
catch(IT_Bus::Exception& e) {
4   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.

CHAPTER 4

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

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 4 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.

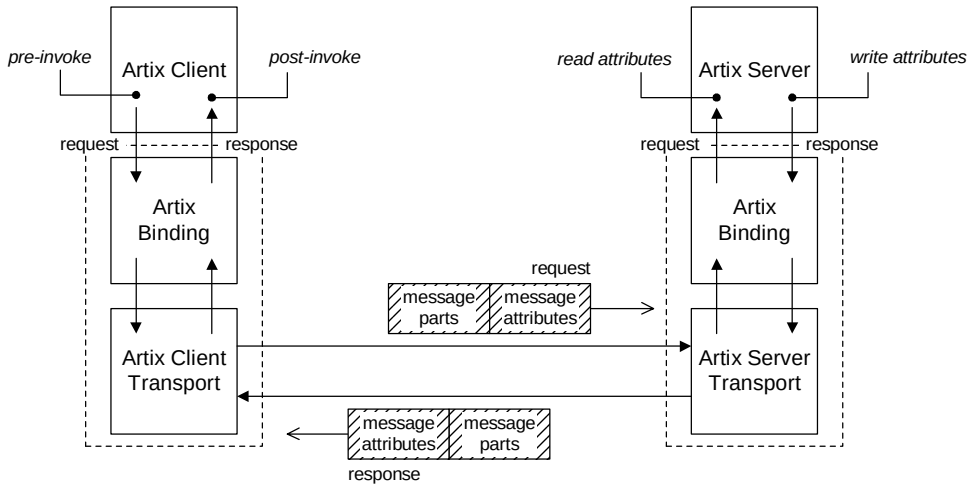

Figure 4: *Passing Message Attributes in Input and Output Messages*

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 73.
- "Transport-Specific API" on page 77.

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 the *Artix User's Guide* 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	*ArtixInstallDir*`/artix/`*Version*`/schemas/http-conf.xsd`
MQ Series	*ArtixInstallDir*`/artix/`*Version*`/schemas/mq.xsd`
Tibco	*ArtixInstallDir*`/artix/`*Version*`/schemas/tibrv.xsd`

HTTP schema example

Example 30 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 30: *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: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 71).

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

- Inheritance hierarchy.
- MessageAttributes class.
- NamedAttributes class.

Inheritance hierarchy

Figure 5 shows the inheritance hierarchy for the classes involved in the name-value API for message attributes.

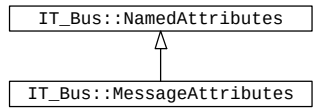

Figure 5: *Inheritance Hierarchy for IT_Bus::MessageAttributes Class*

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 31, 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 31: *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 31: *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 31: *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 6 shows the inheritance hierarchy for the classes involved in the transport-specific API for message attributes.

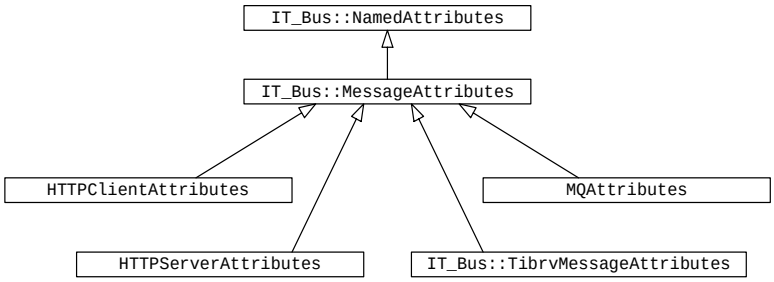

Figure 6: *Inheritance Hierarchy for the Transport-Specific API*

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 32, 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 32: *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();
```

Example 32: *Getting and Setting Tibco Message Attributes*

```
        virtual const Double& get_time_limit();
        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 `IT_Bus::Port` object by calling `get_port()` on the client proxy object.
2	Call the `use_input_message_attributes()` and `use_output_message_attributes()` functions on the `IT_Bus::Port` object to initialize the message attribute functionality.
3	Pre-invoke step—set the input message attributes on the `IT_Bus::Port` object.
4	Invoke a WSDL operation on the client proxy.
5	Post-invoke step—read the output message attributes from the `IT_Bus::Port` object.

C++ example

To use message attributes in a sample client, you can modify the HelloWorld HTTP Soap client as shown in Example 33. Edit the `client.cxx` file, which is located in the *ArtixInstallDir*`/artix/`*Version*`/demos/hellow_world/http_soap/client` directory. In Example 33, 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 33: *Using Message Attributes in a Client*

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

Example 33: *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

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:

Step	Action
1	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.
2	In the servant constructor, obtain a reference to the servant's port and call the `use_input_message_attributes()` and `use_output_message_attributes()` to initialize the message attribute functionality.
3	Within an execution context, obtain an `IT_Bus::Port` object by calling `get_port()` on the server stub base 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 34. Edit the `HelloWorldImpl.cxx` file, which is located in the *ArtixInstallDir*`/artix/`*Version*`/demos/hellow_world/http_soap/server` directory. In Example 34, 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 34: *Using Message Attributes in a Server*

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

IT_USING_NAMESPACE_STD
using namespace IT_Bus;

HelloWorldImpl::HelloWorldImpl(IT_Bus::Bus_ptr bus,
   IT_Bus::Port* port)
  : HelloWorldServer(bus,port)
{
1     get_port().use_input_message_attributes();
    get_port().use_output_message_attributes();
}

void
HelloWorldImpl::sayHi(
    IT_Bus::String & Response
) IT_THROW_DECL((IT_Bus::Exception))
{
    // Read input message attributes.
    IT_Bus::MessageAttributes& hw_input =
2         get_port().get_input_message_attributes();
    try {
3         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;
    }
4     catch (IT_Bus::NoSuchAttributeException) { }

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

Example 34: *Using Message Attributes in a Server*

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

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

The preceding server code example can be explained as follows:

1. The `get_port()` operation is defined on the `IT_Bus::ServerStubBase` class, which is a base class of `HelloWorldImpl`. It returns a reference to the `IT_Bus::Port` object that represents the server connection.

   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.

2. To read the input message attribute object on the server side, call `get_input_message_attributes()` on the server port object.
3. 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.
4. The `IT_Bus::NoSuchAttributeException` exception is thrown here if you try to read an input attribute that was not sent by the client.
5. You can send output message attributes back to the client by setting attributes on the output message attributes object, `hw_output`.

CHAPTER 5

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

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 7 shows an overview of the Artix dynamic configuration mechanism.

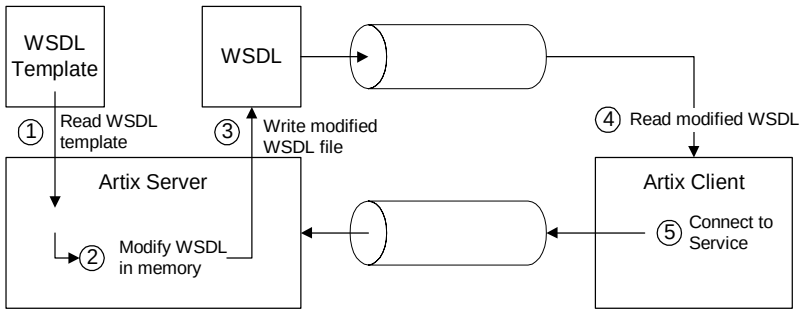

Figure 7: *Dynamic Configuration Mechanism*

Process of dynamic configuration

The process of dynamic configuration shown in Figure 7 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 `<port>` 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 `<port>` 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 90.

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 `IT_Bus::init()` 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 35.

Example 35: *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 35: *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"
        );
    ...
}
```

CHAPTER 6

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

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

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
`xsd:boolean`	`IT_Bus::Boolean`
`xsd:byte`	`IT_Bus::Byte`
`xsd:unsignedByte`	`IT_Bus::UByte`
`xsd:short`	`IT_Bus::Short`
`xsd:unsignedShort`	`IT_Bus::UShort`
`xsd:int`	`IT_Bus::Int`
`xsd:unsignedInt`	`IT_Bus::UInt`
`xsd:long`	`IT_Bus::Long`
`xsd:unsignedLong`	`IT_Bus::ULong`
`xsd:float`	`IT_Bus::Float`
`xsd:double`	`IT_Bus::Double`
`xsd:string`	`IT_Bus::String`
`xsd:dateTime`	`IT_Bus::DateTime`
`xsd:decimal`	`IT_Bus::Decimal`
`xsd:base64Binary`	`IT_Bus::Base64Binary`
`xsd:hexBinary`	`IT_Bus::HexBinary`

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::String`s 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.

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)`
		`void getUTCTimeZoneOffset(`
		`short & hour_offset,`
		`short & minute_offset)`

minute offset from GMT	`short`	

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/fixed_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 36 shows how a server might implement the `echoHexBinary` operation.

Example 36: *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 37 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 37: *WSDL Example of Derivation with the Enumeration Facet*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >
    <types>
        <schema ... >
            <simpleType name="ColorEnum">
                <restriction base="xsd:string">
                    <enumeration value="RED"/>
                    <enumeration value="GREEN"/>
                    <enumeration value="BLUE"/>
                </restriction>
            </simpleType>
            ...
</definitions>
```

C++ mapping of enumeration facet

The WSDL-to-C++ compiler maps the `ColorEnum` restricted type to the `ColorEnum` C++ class, as shown in Example 38. The only values that can legally be set using the `set_value()` member function are the strings `RED`, `GREEN`, or `BLUE`.

Example 38: *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
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

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 *PortTypeName*`Types.h` and *PortTypeName*`Types.cxx` files.

The WSDL-to-C++ mapping defines accessor and modifier functions for each element in the sequence complex type.

Occurrence constraints

Occurrence constraints, which are specified using the `minOccurs` and `maxOccurs` attributes, are supported for sequence complex types. See "Occurrence Constraints" on page 128.

WSDL example

Example 39 shows an example of a sequence, `SequenceType`, with three elements.

Example 39: *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 39) to the SequenceType C++ class. An outline of this class is shown in Example 40.

Example 40: *Mapping of SequenceType to C++*

```
// C++
class SequenceType : public IT_Bus::SequenceComplexType
{
  public:
    SequenceType();
    SequenceType(const SequenceType& copy);
    virtual ~SequenceType();
    ...
    virtual const IT_Bus::QName & get_type() const;

    SequenceType& operator= (const SequenceType& assign);

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

    const IT_Bus::Int &   getvarInt() const;
    IT_Bus::Int &         getvarInt();
    void                  setvarInt(const IT_Bus::Int & val);

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

  private:
    ...
};
```

Each *ElementName* element declared in the sequence complex type is mapped to a pair of accessor/modifier functions, get*ElementName*() and set*ElementName*().

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 41 shows how a client could use a proxy instance, bc, to invoke the echoSequence operation.

Example 41: *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 Sequence 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 *PortTypeName*`Types.h` and *PortTypeName*`Types.cxx` files.

The WSDL-to-C++ mapping defines accessor and modifier functions for each element in the choice complex type. The choice complex type is effectively equivalent to a C++ union, so only one of the elements is accessible at a time. The C++ implementation defines a discriminator, which tells you which of the elements is currently selected.

Occurrence constraints

Occurrence constraints are currently not supported for choice complex types.

WSDL example

Example 42 shows an example of a choice complex type, `ChoiceType`, with three elements.

Example 42: *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 42) to the SequenceType C++ class. An outline of this class is shown in Example 43.

Example 43: *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 43: *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, `get`*ElementName*`()` and `set`*ElementName*`()`.

The member functions have the following effects:

- `set`*ElementName*`()`—select the *ElementName* element, setting the discriminator to the *ElementName* label and initializing the value of *ElementName*.
- `get`*ElementName*`()`—get the value of the *ElementName* element. You should always check the discriminator before calling the `get`*ElementName*`()` accessor. If *ElementName* is not currently selected, the value returned by `get`*ElementName*`()` is undefined.
- `get_discriminator()`—returns the value of the discriminator.

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 44 shows how a client could use a proxy instance, `bc`, to invoke the `echoChoice` operation.

Example 44: *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 44: *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 *PortTypeName*`Types.h` and *PortTypeName*`Types.cxx` files.

The WSDL-to-C++ mapping defines accessor and modifier functions for each element in the all complex type. With an all complex type, the order in which the elements are transmitted is immaterial.

Note: An all complex type can only be declared as the *outermost* group of a complex type. Hence, you cannot nest an all model group, `<all>`, directly inside other model groups, `<all>`, `<sequence>`, or `<choice>`. You may, however, define an all complex type and then declare an element of that type within the scope of another model group.

Occurrence constraints

Occurrence constraints are supported for the elements of XML schema all complex types.

WSDL example

Example 45 shows an example of an all complex type, `AllType`, with three elements.

Example 45: *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 45) to the `AllType` C++ class. An outline of this class is shown in Example 46.

Example 46: *Mapping of AllType to C++*

```
// C++
class AllType : public IT_Bus::AllComplexType
{
  public:
    AllType();
    AllType(const AllType& copy);
    virtual ~AllType();

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

    AllType& operator= (const AllType& assign);

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

    const IT_Bus::Int & getvarInt() const;
    IT_Bus::Int & getvarInt();
    void setvarInt(const IT_Bus::Int & val);

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

  private:
    ...
};
```

Each *ElementName* element declared in the sequence complex type is mapped to a pair of accessor/modifier functions, `get`*ElementName*`()` and `set`*ElementName*`()`.

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 47 shows how a client could use a proxy instance, `bc`, to invoke the `echoAll` operation.

Example 47: *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 48 shows how to define a sequence type with a single attribute, `prop`, of `xsd:string` type.

Example 48: *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 49 shows an outline of the C++ SequenceType class generated from Example 48 on page 119, which defines accessor and modifier functions for the prop attribute.

Example 49: *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 50.

WSDL example

Example 50 shows an example of a nested complex type, which features a choice complex type, `NestedChoiceType`, nested inside a sequence complex type, `SeqOfChoiceType`.

Example 50: *Definition of Nested Complex Type*

```
<schema targetNamespace="http://soapinterop.org/xsd"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <complexType name="NestedChoiceType">
        <choice>
            <element name="varFloat" type="xsd:float"/>
            <element name="varInt" type="xsd:int"/>
        </choice>
    </complexType>
    <complexType name="SeqOfChoiceType">
        <sequence>
         <element name="varString" type="xsd:string"/>
         <element name="varChoice" type="xsd1:NestedChoiceType"/>
        </sequence>
    </complexType>
    ...
</schema>
```

C++ mapping of NestedChoiceType

The XML schema choice complex type, `NestedChoiceType`, is a simple choice complex type, which is mapped to C++ in the standard way. Example 51 shows an outline of the generated C++ `NestedChoiceType` class.

Example 51: *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 52 shows an outline of the generated C++ `SeqOfChoiceType` class, which shows how the nested complex type is mapped within a sequence complex type.

Example 52: *Mapping of SeqOfChoiceType to C++*

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

Example 52: *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 47 shows how a client could use a proxy instance, `bc`, to invoke the `echoSeqOfChoice` operation.

Example 53: *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 53: *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 54 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 54: *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 55 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 55: *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 56 shows an outline of the C++ `internationalPrice` class generated from Example 55 on page 126.

Example 56: *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 56: *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 132.

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. For most purposes, you can treat the element list types as if they were `IT_Vector` (see "IT_Vector Template Class" on page 150). 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 57 shows the definition of a sequence type, SequenceType, which contains a list of integer elements followed by a list of string elements.

Example 57: *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 58 shows an outline of the C++ SequenceType class generated from Example 57 on page 129, which defines accessor and modifier functions for the varInt and varString elements.

Example 58: *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, &m_varInt_qname, 1,
   100> & getvarInt() const;
```

Example 58: *Mapping of SequenceType to C++*

```
    IT_Bus::ElementListT<IT_Bus::Int, &m_varInt_qname, 1, 100> &
   getvarInt();

    void setvarInt(const IT_Bus::ElementListT<IT_Bus::Int,
   &m_varInt_qname, 1, 100> & val);

    const IT_Bus::ElementListT<IT_Bus::String,
   &m_varString_qname, 0, -1> & getvarString() const;

    IT_Bus::ElementListT<IT_Bus::String, &m_varString_qname, 0,
   -1> & getvarString();

    void setvarString(const IT_Bus::ElementListT<IT_Bus::String,
   &m_varString_qname, 0, -1> & val);

  private:
    ...
};
```

Because the `IT_Bus::ElementListT` template supports automatic conversion to `IT_Vector`, you can 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 150.
- 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 138.

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 59 shows how to define a one-dimensional string array, `ArrayOfString`, whose size can lie anywhere in the range `0` to unbounded.

Example 59: *Definition of an Array of Strings*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >
    <types>
        <schema ... >
            <complexType name="ArrayOfString">
                <sequence>
                    <element name="varString" type="xsd:string"
                            minOccurs="0" maxOccurs="unbounded"/>
                </sequence>
            </complexType>
            ...
    ...
</definitions>
```

C++ mapping

Example 60 shows how the `ArrayOfString` string array (from Example 59 on page 133) maps to C++.

Example 60: *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,
   &ArrayOfString_varString_qname, 0, -1> & getvarString()
   const;
```

Example 60: *Mapping of ArrayOfString to C++*

```
    IT_Bus::ElementListT<IT_Bus::String,
   &ArrayOfString_varString_qname, 0, -1> & getvarString();

    void setvarString(const IT_Bus::ElementListT<IT_Bus::String,
   &ArrayOfString_varString_qname, 0, -1> & 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 128). 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 61 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 154).

Example 61: *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 121 for a discussion of nested types). Example 62 shows an example of how to define a two-dimensional string array, `ArrayOfArrayOfString`.

Example 62: *Definition of a Multi-Dimensional String Array*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >
    <types>
        <schema ... >
            <complexType name="ArrayOfString">
                <sequence>
                    <element name="varString" type="xsd:string"
                             minOccurs="0" maxOccurs="unbounded"/>
                </sequence>
            </complexType>
            <complexType name="ArrayOfArrayOfString">
                <sequence>
                    <element name="nestArray"
                              type="xsd1:ArrayOfString"
                             minOccurs="0" maxOccurs="unbounded"/>
                </sequence>
            </complexType>
            ...
    ...
</definitions>
```

Both the nested array type, `ArrayOfArrayOfString`, and the sub-array type, `ArrayOfString`, must conform to the standard array definition syntax. Multi-dimensional arrays can be nested to an arbitrary degree, but each sub-array must be a named type (that is, anonymous nested array types are not supported).

C++ example for multidimensional array

Example 63 shows an example of how to allocate and initialize a multi-dimensional array, of `ArrayOfArrayOfString` type.

Example 63: *C++ Example for a Multi-Dimensional Array*

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

Example 63: *C++ Example for a Multi-Dimensional Array*

```
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 64 shows how an instance, `a2`, of `ArrayOfArrayOfString` type converts to an instance of `IT_Vector<`*ArrayOfString*`>` type by assignment.

Example 64: *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 150.
- The section on C++ ANSI vectors in *The C++ Programming Language*, third edition, by Bjarne Stroustrup.

SOAP Arrays

Overview

In addition to the basic array types described in "Arrays" on page 132, 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

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 65 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 65: *Definition of the ArrayOfSOAPString SOAP Array*

```
<definitions name="BaseService"
   targetNamespace="http://soapinterop.org/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://soapinterop.org/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://soapinterop.org/xsd">
    <types>
        <schema targetNamespace="http://soapinterop.org/xsd"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
            <complexType name="ArrayOfSOAPString">
                <complexContent>
                    <restriction base="SOAP-ENC:Array">
                        <attribute ref="SOAP-ENC:arrayType"
                            wsdl:arrayType="xsd:string[]"/>
                    </restriction>
                </complexContent>
            </complexType>
            ...
</definitions>
```

Sample encoding

Example 66 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 66: *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 67 shows a C++ example of how to allocate and initialize an `ArrayOfSOAPString` instance with four elements.

Example 67: *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 143.
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 146). 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 68 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 68: *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 69 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 69: *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 70 shows a C++ example of how to allocate and initialize an `Array2OfInt` instance with dimensions, `[2,3]`.

Example 70: *Initializing an Array2OfInt SOAP Array*

```
// C++
const size_t extents2[] = {2, 3};
Array2OfInt a2_soap(extents2);

size_t position[2];
size_t i_max = a2_soap.get_extents()[0];
size_t j_max = a2_soap.get_extents()[1];
for (size_t i=0; i<i_max; i++) {
    position[0] = i;
    for (size_t j=0; j<j_max; j++) {
        position[1] = j;
        a2_soap[position] = (IT_Bus::Int) (i+1)*(j+1);
    }
```

Example 70: *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 71 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 71: *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 72 shows an example of how to initialize a sparse array of `Array2OfInt` type.

Example 72: *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 70 on page 144). 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 73 shows an example of how to read a sparse array of `Array2OfInt` type.

Example 73: *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 74 shows the encoding of a partially transmitted `ArrayOfSOAPString` instance.

Example 74: *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

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 9 are *not* defined in `IT_Vector`.

Table 9: *Member Functions Not Defined in IT_Vector*

Function	Type of Operation
`at()`	Element access (with range check)
`clear()`	List operation
`assign()`	Assignment
`resize()`	Size and capacity
`max_size()`	

Although `clear()` is not defined, you can easily get the same effect for a vector, `v`, by calling `erase()` as follows:

```
v.erase(v.begin(), v.end());
```

This has the effect of erasing all the elements in `v`, leaving an array of size `0`.

Basic usage of IT_Vector

The `size()` member function and the indexing operator `[]` is all that you need to perform basic manipulation of vectors. Example 75 shows how to use these basic vector operations to initialize an integer vector with the first one hundred integer squares.

Example 75: *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 75 on page 152 can be written in a more idiomatic style using vector iterators, as shown in Example 76.

Example 76: *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 10 lists the member types defined in `IT_Vector<T>`.

Table 10: *Member Types Defined in IT_Vector<T>*

Member Type	Description
`value_type`	Type of element.
`size_type`	Type of subscripts.
`difference_type`	Type of difference between iterators.
`iterator`	Behaves like `value_type*`.
`const_iterator`	Behaves like `const value_type*`.
`reverse_iterator`	Iterates in reverse, like `value_type*`.
`const_reverse_iterator`	Iterates in reverse, like `const value_type*`.
`reference`	Behaves like `value_type&`.
`const_reference`	Behaves like `const value_type&`.

Iterators

Table 11 lists the `IT_Vector` member functions returning iterators.

Table 11: *Iterator Member Functions of IT_Vector<T>*

Iterator Member Function	Description
`begin()`	Points to first element.
`end()`	Points to last element.
`rbegin()`	Points to first element of reverse sequence.
`rend()`	Points to last element of reverse sequence.

Element access

Table 12 lists the `IT_Vector` element access operations.

Table 12: *Element Access Operations for IT_Vector<T>*

Element Access Operation	Description
`[]`	Subscripting, unchecked access.
`front()`	First element.
`back()`	Last element.

Stack operations

Table 13 lists the `IT_Vector` stack operations.

Table 13: *Stack Operations for IT_Vector<T>*

Stack Operation	Description
`push_back()`	Add to end.
`pop_back()`	Remove last element.

List operations

Table 14 lists the `IT_Vector` list operations.

Table 14: *List Operations for IT_Vector<T>*

List Operations	Description
`insert(p,x)`	Add `x` before `p`.
`insert(p,n,x)`	Add `n` copies of `x` before `p`.
`insert(first,last)`	Add elements from `[first:last[` before `p`.
`erase(p)`	Remove element at `p`.
`erase(first,last)`	Erase `[first:last[`.

Other operations

Table 15 lists the other operations supported by `IT_Vector`.

Table 15: *Other Operations for IT_Vector<T>*

Operation	Description
`size()`	Number of elements.
`empty()`	Is the container empty?
`capacity()`	Space allocated.
`reserve()`	Reserve space for future expansion.
`swap()`	Swap all the elements between two vectors.
`==`	Test vectors for equality (member-wise).

CHAPTER 7

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

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 -w 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). This mapping is used, for example, by the Orbix ASP product.

These alternative approaches are illustrated in Figure 8.

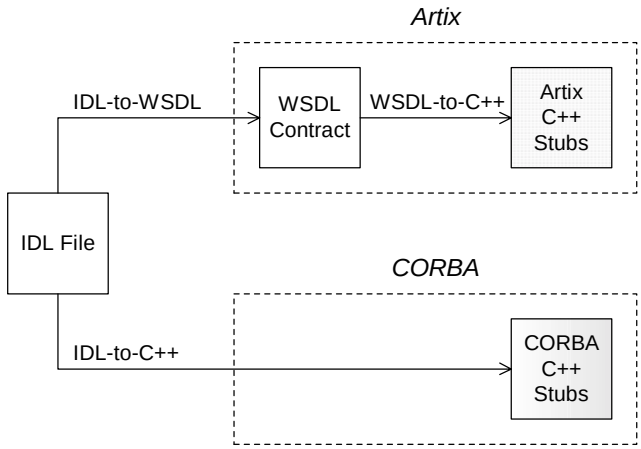

Figure 8: *Artix and CORBA Alternatives for IDL to C++ Mapping*

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:

- `any`.
- `wchar`.
- `wstring`.
- `long double`.
- Object references.
- Value types.
- Boxed values.
- Local interfaces.
- Abstract interfaces.
- forward-declared interfaces.

IDL Basic Type Mapping

Overview

Table 16 shows how IDL basic types are mapped to WSDL and then to C++.

Table 16: *Artix Mapping of IDL Basic Types to C++*

IDL Type	WSDL Schema Type	C++ Type
`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`	*Not supported*	*Not supported*
`octet`	`xsd:unsignedByte`	`IT_Bus::UByte`
`fixed`	`xsd:decimal`	`IT_Bus::Decimal`
`Object`	*Not supported*	*Not supported*
`any`	*Not supported*	*Not supported*

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 104.

---

union type

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

```
// IDL
module SampleTypes {
    union Poly switch(short) {
        case 1: short theShort;
        case 2: string theString;
    };
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::Poly` union to an XML schema choice complex type, `SampleTypes.Poly`, as follows:

```
<xsd:complexType name="SampleTypes.Poly">
    <xsd:choice>
        <xsd:element name="theShort" type="xsd:short"/>
        <xsd:element name="theString" type="xsd:string"/>
    </xsd:choice>
</xsd:complexType>
```

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

```
// C++
class SampleTypes_Poly : public IT_Bus::ChoiceComplexType
{
  public:
    ...
    const IT_Bus::Short gettheShort() const;
    void settheShort(const IT_Bus::Short& val);

    const IT_Bus::String& gettheString() const;
    void settheString(const IT_Bus::String& val);

    enum PolyDiscriminator
    {
        theShort,
        theString,
        Poly_MAXLONG=-1L
    } m_discriminator;

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

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

Programming with the Union Type

For details of how to use the union type in C++, see "Choice Sequence Types" on page 112.

struct type

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

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

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

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

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

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

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

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

Programming with the Struct Type

For details of how to use the struct type in C++, see "Sequence Complex Types" on page 109.

---

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 132 and "IT_Vector Template Class" on page 150.

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 occurence constraints, `SampleTypes.ArrOfStruct`, as follows:

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

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

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

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

Programming with Array Types

For details of how to use array types in C++, see "Arrays" on page 132 and "IT_Vector Template Class" on page 150.

---

exception types

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

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

The IDL-to-WSDL compiler maps the `SampleTypes::GenericExc` exception to a WSDL sequence type, `SampleTypes.GenericExc`, and to a WSDL fault message, `_exception.SampleTypes.GenericExc`, as follows:

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

The WSDL-to-C++ compiler maps the `SampleTypes.GenericExc` and `_exception.SampleTypes.GenericExc` types to C++ classes, `SampleTypes_GenericExc` and `_exception_SampleTypes_GenericExc`, as follows:

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

Programming with Exceptions in Artix

For an example of how to initialize, throw and catch a WSDL fault exception, see "Propagating Exceptions" on page 27.

---

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 a sequence or an 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.
- 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.

Operation mapping

Example 77 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 77: *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 77 on page 171, map to C++ as shown in Example 78,

Example 78: *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 79 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 79: *Example IDL Attributes*

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

The attributes from the preceding IDL, Example 79 on page 172, map to C++ as shown in Example 80,

Example 80: *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

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X