Developing Artix Applications in C++

Version 2.0, March 2004

Making Software Work Together™

Updated: 13-Jul-2004

M 3 1 9 1

Contents

List of Tables

Preface

What is Covered in this Book

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

Who Should Read this Book

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

Organization of this guide

This guide is divided as follows:

Part I <PART-TITLE>

<PART-DESCRIPTION>

Related Documentation

The Artix library includes the following books:

- *Getting Started with Artix*
- *Deploying and Managing Artix Solutions*
- *Designing Artix Solutions from the Command Line*
- *Designing Artix Solutions using Artix Designer*
- *Developing Artix Applications in C++*
- *Developing Artix Applications in Java*
- *Artix Security Guide*
- *Artix Tutorial Guide*

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

Online Help

Artix includes comprehensive online help, providing:

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

The **Help** menu in Artix Designer provides access to this online help.

Suggested Path for Further Reading

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

1. *Getting Started with Artix Encompass*

   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++ Web Service.

2. *Artix Tutorial*

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

3. *Deploying and Managing Artix Solutions*

   The deployment guide describes deploying Artix enabled systems. It provides detailed examples for a number of typical use cases.

4. *Designing Artix Solutions with Artix Designer*

   The Artix Designer book describes how to use the Artix GUI to describe your services in an Artix contract.

5. *Developing Artix Applications in C++/Java*

   The development guide discusses the technical aspects of programming applications using the Artix API.

6. *Designing Artix Solutions from the Command Line*

This book provides detailed information about the WSDL extensions used in Artix contracts and explains the mappings between data types and Artix bindings.

Additional Resources for Information

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

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 book 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 book uses 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 a number of stub files from the Artix contract. They are named according to the port type name, *PortTypeName*, specified in the logical portion of the Artix contract. If the contract specifies more than one port type, code will be generated for each one.

The following stub files are generated:

*PortTypeName***.h** defines the superclass from which the client and server are implemented. It represents the API used by the service defined in the contract.

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

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

*PortTypeName***_wsdlTypes.h and** *PortTypeName***_wsdlTypes.cxx** define the complex datatypes defined in the contract (if any).

*PortTypeName***_wsdlTypesFactory.h and** *PortTypeName***_wsdlTypesFactory.cxx** define factory classes for the complex datatypes defined in the contract (if any).

Generating code from the command line

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

```
wsdltocpp [options] { WSDL-URL | SCHEMA-URL }
   [-e web_service_name] [-t port] [-b binding_name]
   [-i port_type] [-d output-dir] [-n namespace]
    [-nimport namespace] [-impl [-m {NMAKE | UNIX} ] | -jp
   plugin_class] [-f] [-server] [-client] [-sample] [-plugin]
   [-v] [-license] [-declspec declspec] [-all] [-?] [-flags]
   [-upper|-lower|-minimal|-mapper class]
```

You must specify the location of a valid WSDL contract file, *WSDL_URL*, for the code generator to work. You can also supply the following optional parameters:

`-i` *port_type*	Specifies the name of the port type for which the tool will generate code. The default is to use the first port type listed in the contract.
`-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.
`-m {NMAKE | UNIX}`	Used in combination with `-impl` to generate a makefile for the specified platform (`NMAKE` for Windows or `UNIX` for UNIX). For example, the options, `-impl -m NMAKE`, would generate a Windows makefile.

`-f`	*Deprecated*—No longer used (was needed to support routing in earlier versions.
`-server`	Generates code for a sample implementation of a server.
`-client`	Generates code for a sample implementation of a client.
`-sample`	Generates code for a sample implementation of a client and a server (equivalent to `-server -client`).
`-plugin`	Generates servant registration code as a Bus plug-in. See "Customizing servant registration" on page 19 for details.
`-v`	Displays the version of the tool.
`-license`	Displays the currently available licenses.
`-declspec` *`declspec`*	Creates NT declaration specifiers for `dllexport` and `dllimport`. This option makes it easier to package Artix stubs in a DLL library. See "Building Artix Stub Libraries on Windows" on page 24 for details.
`-all`	Generate stub code for all of the port types and the types that they use. This option is useful when multiple port types are defined in a WSDL contract.
`-?`	Displays help on using the command line tool.
`-flags`	Displays detailed information about the options.
`-nimport` *`namespace`*	Specifies the namespace under which code from imported schema is generated. If *`namespace`* is left blank, the code for the imported schema will be generated in the global namespace.

C++ Namespaces

Artix namespaces

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

Solution specific namespaces

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

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

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

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

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

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

Defining a WSDL Interface

Overview

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

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

Restrictions

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

- Some simple atomic types are not supported—see "Unsupported Simple Types" on page 227.

WSDL example

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

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

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

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

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

Developing a Server

Overview

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

Generating the server implementation class

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

Generated code

The implementation class code consists of two files:

*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");
}
```

Writing the server main()

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

Initializing the Bus

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

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

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

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

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

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

Registering the Servant Objects

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

Example 4: *Registering a Servant Object for HelloWorld*

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

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

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

Running the Bus

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

Shutting the Bus down

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

Completed server main()

Example 5 on page 11 shows how the `main()` for the server defined by the HelloWorld contract might look.

Example 5: *ConverterServer main()*

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

IT_USING_NAMESPACE_STD
using namespace IT_Bus;

int main(int argc, char* argv[])
{
    try {
        IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);

        HelloWorldImpl servant(bus);

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

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

        IT_Bus::run();
    }
    catch (IT_Bus::Exception& e)
    {
      cout << "Error occurred: " << e.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 constructors as shown in Example 6 on page 12.

Example 6: *Generated Client Constructors*

```
HelloWorldClient();

HelloWorldClient(const IT_Bus::String & wsdl);

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

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

Constructor with no arguments

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

Constructor with WSDL URL argument

The second constructor takes one argument that allows you to specify the URL of the contract defining the client's behavior. The client uses the port and service specified at code generation time using the `-t` and `-b` flags. This is useful for situations where the contracts are stored in a central location.

Constructor with three arguments

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

`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 7 on page 13, you could instantiate the client proxy to use either HTTP or Tuxedo.

Example 7: *Multiple Ports Defined for HelloWorld*

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

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

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

Constructor with a reference argument

The fourth constructor takes one argument representing an Artix reference, IT_Bus::Reference. The Artix reference contains complete service and port details, including addressing information, enabling the client proxy to open a connection to a remote service. For a detailed discussion of Artix references, see "Artix References" on page 75.

Invoking the operations

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

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

Shutting the bus down

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

Full client code

A client developed to access the service defined by the HelloWorldService contract will look similar to Example 8.

Example 8: *HelloWorld Client*

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

Example 8: *HelloWorld Client*

```
#include <it_cal/iostream.h>

1   #include "HelloWorldClient.h"

IT_USING_NAMESPACE_STD
using namespace IT_Bus;

using namespace HW;

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

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

        String string_in;
        String string_out;

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

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

    return 0;
}
```

The code does the following:

1. The *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`.

Generating a Sample Application from WSDL

Overview

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

Sample WSDL file

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

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

Generating the sample application

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

Windows

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

UNIX

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

Generated files

The preceding `wsdltocpp` command generates the following files:

Stub Files

PortType`.h`
PortType`Client.h`
PortType`Server.h`
PortType`Client.cxx`
PortType`Server.cxx`

Client Implementation Files

PortType`ClientSample.cxx`

Server Implementation Files

PortType`ServerSample.cxx`
PortType`Impl.cxx`
PortType`ServantBusPlugIn.cxx`

Makefile

`Makefile`

Building the sample application

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

Windows

```
> nmake -all
```

UNIX

```
% make -all
```

Customizing the servant implementation

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

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

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

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


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

GreeterImpl::~GreeterImpl()
{
}

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

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

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

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

Customizing servant registration

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

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

Example 10: *Extract from the GreeterServantBusPlugIn Class*

```
// C++
...
GreeterServantBusPlugIn::GreeterServantBusPlugIn(
    Bus_ptr bus
) IT_THROW_DECL((Exception))
  :
    BusPlugIn(bus),
    m_servant(bus),
    m_service_qname("", "SOAPService",
   "http://www.iona.com/hello_world_soap_http")
{
    // complete
}
```

Example 10: *Extract from the GreeterServantBusPlugIn Class*

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

void
GreeterServantBusPlugIn::bus_init(
) IT_THROW_DECL((Exception))
{
    get_bus()->register_servant(
        m_servant,
        "hello_world.wsdl",
        m_service_qname
    );
}
...
```

If you want to change the details of servant registration, you can edit the `register_servant()` calls in the `GreeterServantBusPlugIn.cxx` file. For a detailed discussion of servant registration, see "Registering Servants" on page 50.

Automatic plug-in activation

In order to have any effect, an Artix plug-in must register itself with the Artix Bus and the Bus must be configured to activate the plug-in. In the case of the generated plug-in class, however, registration and activation of the plug-in occur automatically.

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

```
// C++
GlobalBusORBPlugIn bus_plugin(
    "SOAPService@http://www.iona.com/hello_world_soap_http",
    plugin_factory
);
```

The `GlobalBusORBPlugIn` is an object that automatically registers and activates the plug-in (whose name is given by the string `SOAPService@http://www.iona.com/hello_world_soap_http`). In contrast

to regular plug-in objects (of `BusORBPlugIn` type), it is *not* necessary to activate the plug-in by adding the plug-in name to the `orb_plugins` list; activation of `GlobalBusORBPlugIn` objects is automatic.

Compiling and Linking an Artix Application

Compiler Requirements

An application built using Artix requires a number of IONA-supplied C++ header files in order to compile. The directory containing these include files must be added to the include path for the compiler, so that when the compiler processes the generated files, it is able to find the necessary included infrastructure header files.

The following include path directives should be given to the compiler:

```
-I"$(IT_PRODUCT_DIR)\artix\$(IT_PRODUCT_VER)\include"
```

Linker Requirements

A number of Artix libraries are required to link with an application built using Artix. The following directives should be given to the linker:

```
-L"$(IT_PRODUCT_DIR)\artix\$(IT_PRODUCT_VER)\lib" it_bus.lib it_afc.lib it_art.lib it_ifc.lib
```

Table 1 shows the libraries that are required for linking an Artix application and their function.

Table 1: *Artix Import Libraries for Linking with an Application*

Windows Libraries	UNIX Libraries	Description

`it_bus.lib`	`libit_bus.so`
	`libit_bus.sl`
	`libit_bus.a`
		The Bus library provides the functionality required to access the Artix bus. Required for all applications that use Artix functionality.

`it_afc.lib`	`libit_afc.so`
	`libit_afc.sl`
	`libit_afc.a`
		The Artix foundation classes provide Artix specific data type extensions such as `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.

Building Artix Stub Libraries on Windows

Overview

The Artix WSDL-to-C++ compiler features an option, `-declspec`, that simplifies the process of building Dynamic Linking Libraries (DLLs) on the Windows platform. The `-declspec` option defines a macro that automatically inserts export declarations into the stub header files.

Generating stubs with declaration specifiers

To generate Artix stubs with declaration specifiers, use the `-declspec` option to the WSDL-to-C++ compiler, as follows:

```
wsdltocpp -declspec MY_DECL_SPEC BaseService.wsdl
```

In this example, the `-declspec` option would add the following preprocessor macro definition to the top of the generated header files:

```
#if !defined(MY_DECL_SPEC)
#if defined(MY_DECL_SPEC_EXPORT)
#define MY_DECL_SPEC    IT_DECLSPEC_EXPORT
#else
#define MY_DECL_SPEC    IT_DECLSPEC_IMPORT
#endif
#endif
```

Where the `IT_DECLSPEC_EXPORT` macro is defined as `_declspec(dllexport)` and the `IT_DECLSPEC_IMPORT` macro is `_declspec(dllimport)`.

Each class in the header file is declared as follows:

```
class MY_DECL_SPEC ClassName { ... };
```

Compiling stubs with declaration specifiers

If you are about to package your stubs in a DLL library, compile your C++ stub files, *StubFile*`.cxx`, with a command like the following:

`cl -DMY_DECLSPEC_EXPORT ...` *StubFile*`.cxx`

By setting the `MY_DECLSPEC_EXPORT` macro on the command line, `_declspec(dllexport)` declarations are inserted in front of the public class declarations in the stub. This ensures that applications will be able to import the public definitions from the stub DLL.

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 11 on page 26).
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 11 on page 26).
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 11 shows an example of a simple operation, `testParams`, which takes two input parameters, `inInt` and `inoutInt`, and two output parameters, `inoutInt` and `outFloat`.

Example 11: *WSDL Declaration of the testParams Operation*

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

Example 11: *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 12 shows how the preceding WSDL `testParams` operation (from Example 11 on page 26) maps to C++.

Example 12: *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 13 shows an example of an operation, `testReverseParams`, whose parameters are listed in the opposite order to that of the preceding `testParams` operation.

Example 13: *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 14 shows how the preceding WSDL `testReverseParams` operation (from Example 13 on page 28) maps to C++.

Example 14: *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 15 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 15: *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 15: *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 16 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 16: *Raising the SampleFault Fault in the Server*

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

Example 16: *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 17 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 17: *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 17: *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 46 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 18 shows an example of a WSDL operation, `testSeqParams`, with three parameters, `inSeq`, `inoutSeq`, and `outSeq`, of sequence type, `xsd1:SequenceType`.

Example 18: *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 18: *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 19 shows how to allocate, initialize, and deallocate parameters when calling the `testSeqParams` operation.

Example 19: *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 19: *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 20 shows how the parameters are used on the server side, in the C++ implementation of the `testSeqParams` operation.

Example 20: *Server Calling the testSeqParams Operation*

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

Example 20: *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 43 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 18 on page 38 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 46.

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

Example 21: *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 21: *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 22 shows how to call the `testSeqParams` operation using parameters that are allocated on the heap and referenced by *smart pointers*

Example 22: *Client Calling testSeqParams Using Smart Pointers*

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

    BaseClient bc;

    // Allocate all parameters
```

Example 22: *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.

Registering Servants

Overview

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

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

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

In this section

This section contains the following subsections:

Registering a Static Servant

Overview

Initially, when a servant object is created, it is associated with a particular *logical contract* (that is, WSDL port type)[1], but has no association with any *physical contract* (that is, WSDL service). The link between a servant instance and a physical contract must be established explicitly by *registering* the servant.

Figure 1 illustrates the effect of registering a static servant: registration establishes an association between a servant instance and a part of the WSDL model that represents a particular WSDL service.

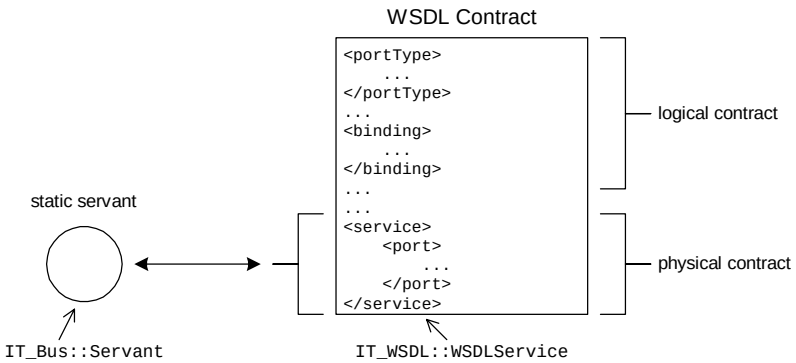

Figure 1: *Relationship between a Static Servant and a WSDL Contract*

Static servant

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

1. Strictly speaking, this is not always the case. Advanced Artix applications could associate a single servant class with multiple port types by overriding the servant dispatch() function.

IT_Bus::Bus registration functions

The `IT_Bus::Bus` class defines the functions in Example 23 to manage the registration of static servants:

Example 23: *The IT_Bus::Bus Static Servant Registration API*

```
// C++
IT_Bus::Service &
register_servant(
    IT_Bus::Servant & servant,
    IT_WSDL::WSDLService & wsdl_service,
    const IT_Bus::String & port_name = ""
) IT_THROW_DECL((IT_Bus::Exception)) = 0;

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

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

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

IT_Bus::Service *
get_service(
    const IT_Bus::QName & service_name
);

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

IT_Bus::Service register_servant() function

In addition to the `IT_Bus::Bus` registration functions, the `IT_Bus::Service` class also supports a `register_servant()` function. The `IT_Bus::Service::register_servant()` function enables you to activate ports individually. This contrasts with the `IT_Bus::Bus::register_servant()` function, which activates all of the ports simultaneously.

Example 24: *The IT_Bus::Service register_servant() Function*

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

Activating single or multiple ports

There are two different styles of programming servant registration, depending on whether you want to activate ports individually or all together, as follows:

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

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

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

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

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

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

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

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

Default threading model

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

See "Servant Threading Models" on page 65 for more information.

Static servant example

Example 25 shows an example (taken from `demos/uncategorized/transient_servants`) which shows how to register a servant as a static servant.

Example 25: *Registering a Static Servant*

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

1   BankImpl my_bank(bus);

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

Example 25: *Registering a Static Servant*

```
3       bus->register_servant(
            my_bank,
            "../wsdl/bank.wsdl",
            service_name
        );

4       IT_Bus::run();

5       bus->remove_service(service_name);
    }
    catch (IT_Bus::Exception& e) { ... }
```

The preceding code example can be explained as follows:

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

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

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

Registering a Transient Servant

Overview

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

For example, consider the WSDL contract for the `demos/uncategorized/transient_servant` demonstration, which has a `Bank` port type and an `Account` port type. If each customer's bank account maps to a service, it is clear that you require an unlimited number of services to represent customer accounts.

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

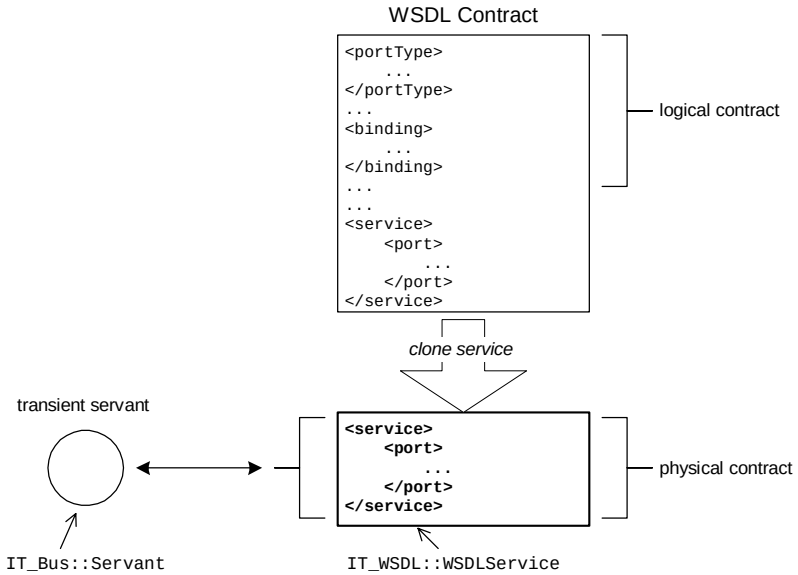

Figure 2: *Relationship between a Transient Servant and a WSDL Contract*

Transient servant

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

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

Reuse of IP ports

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

IT_Bus::Bus transient registration functions

The `IT_Bus::Bus` class defines the functions in Example 26 to manage the registration of transient servants.

Example 26: *The IT_Bus::Bus Transient Servant Registration API*

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

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

Example 26: *The IT_Bus::Bus Transient Servant Registration API*

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

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

IT_Bus::Service *
get_service(
    const IT_Bus::QName & service_name
);

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

IT_Bus::Service register_servant() function

In addition to the `IT_Bus::Bus` registration functions, the `IT_Bus::Service` class also supports a `register_servant()` function. The `IT_Bus::Service::register_servant()` function enables you to activate ports individually. This contrasts with the `IT_Bus::Bus::register_transient_servant()` function, which activates all of the ports simultaneously.

Example 27: *The IT_Bus::Service register_servant() Function*

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

Activating single or multiple ports

There are two different styles of programming transient servant registration, depending on whether you want to activate ports individually or all together, as follows:

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

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

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

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

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

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

bus->register_transient_servant(
    account_servant,
    "bank.wsdl",
    service_name
);
```

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

Default threading model

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

See "Servant Threading Models" on page 65 for more information.

Transient servant example

Example 28 shows a sample implementation of the `Bank` port type's `create_account` operation (taken from `demos/uncategorized/transient_servants`) which shows how to register a servant as a transient servant.

Example 28: *Registering a Transient Servant*

```
// C++
...
1  const IT_Bus::QName AccountImpl::SERVICE_NAME("",
      "AccountService", "http://www.iona.com/bus/demos/bank");
...
void
BankImpl::create_account(
    const IT_Bus::String &account_name,
    IT_Bus::Reference &account_reference
) IT_THROW_DECL((IT_Bus::Exception))
{
    AccountMap::iterator account_iter = m_account_map.find(
                                            account_name
                                        );
    if (account_iter == m_account_map.end())
    {
        cout << "Creating new account: "
             << account_name.c_str() << endl;

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

        // Now put the details for the account into the map so
```

Example 28: *Registering a Transient Servant*

```
        // we can retrieve it later.
        //
        AccountDetails details;
        details.m_service = &service;
        details.m_account = new_account;

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

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

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

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

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

Multi-Threading

Overview

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

In this section

This section contains the following subsections:

Client Threading Issues

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 29 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 29: *Single Client Proxy in Two Threads*

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

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

BaseClient g_bc;
IT_Mutex  mutexBC;

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

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

Two client proxies in two threads

Example 30 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 30: *Two Client Proxies in Two Threads*

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


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

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

Servant Threading Models

Overview

Artix supports a variety of different threading models on the server side. The threading model that applies to a particular service can be specified by programming (see "Setting the Servant Threading Model" on page 68). This subsection provides an overview of each of the servant threading models in Artix, as follows:

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

Default threading model

The default threading model is multi-threaded.

Multi-threaded

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

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

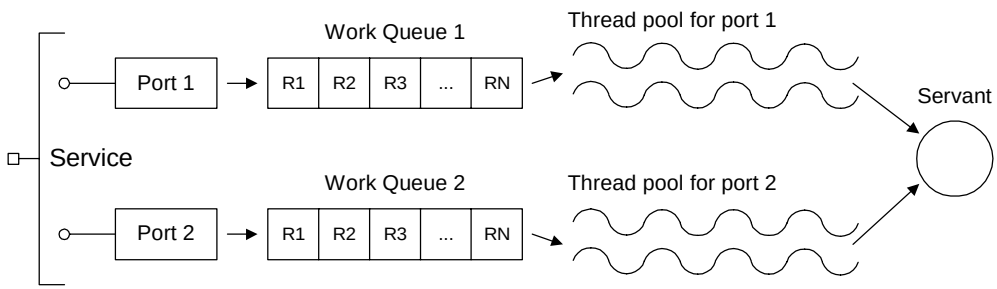

Figure 3: *Outline of the Multi-Threaded Threading Model*

Serialized

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

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

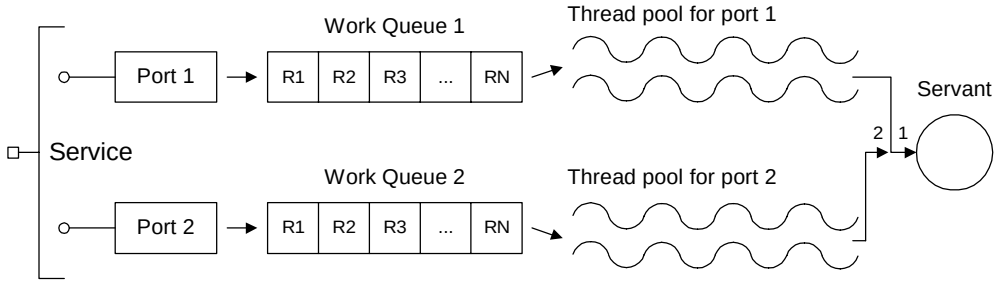

Figure 4: *Outline of the Serialized Threading Model*

Per-port

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

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

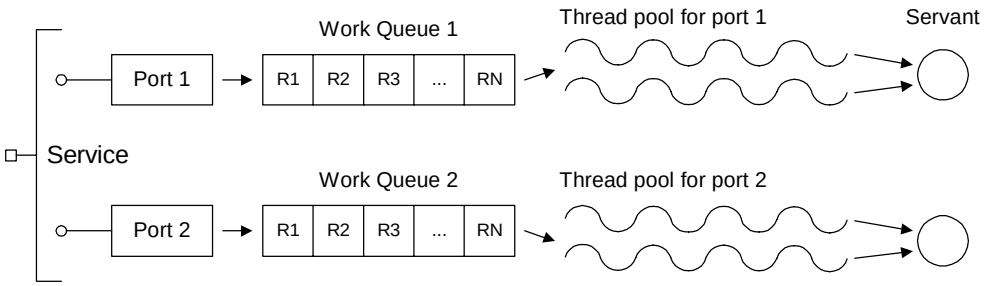

Figure 5: *Outline of the Per-Port Threading Model*

PerThread

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

Figure 6 shows an outline of the `PerThread` threading model. An Artix service can have multiple ports, and each of the ports is served by a work queue that stores the incoming requests. A pool of threads is reserved for each port, and each thread in the pool is associated with a distinct servant instance.

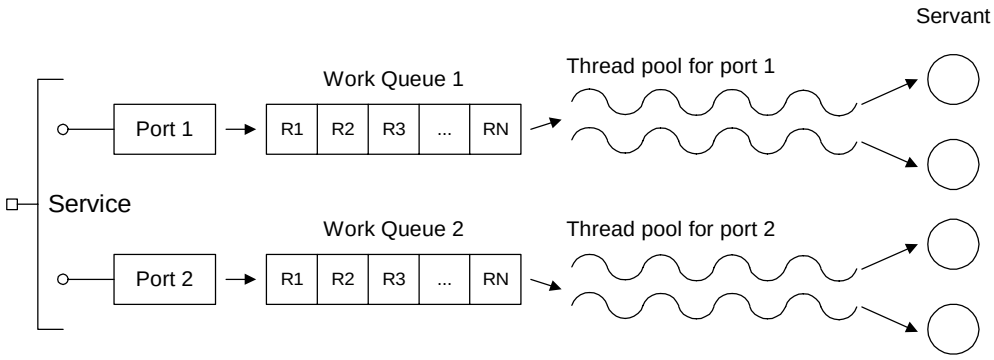

Figure 6: *Outline of the PerThread Threading Model*

PerInvocation

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

The relationship between threads and servants is similar to the case of the `PerThread` threading model (see Figure 6 on page 67). There is a difference in servant lifecycle management, however. Each thread is associated with a servant for the duration of an operation invocation. At the end of the invocation, the servant instance is destroyed.

Setting the Servant Threading Model

Overview

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

How to set a per-port threading model

The per-port threading model can be enabled by employing the two-step style of servant registration (see page 53 and see page 59). For example, you could register distinct servants, `corba_servant` and `soap_servant`, against distinct ports, `CORBAPort` and `SOAPPort`, using the following code example:

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

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

Wrapper servants

The only wrapper servant function that you need is a constructor. Example 31 shows the constructors for each of the wrapper servant classes.

Example 31: *Constructors for the Wrapper Servant Classes*

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

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

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

How to set a threading model using wrapper servants

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

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

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

If you intend to use a `PerThread` or `PerInvocation` threading model, you must implement the `clone()` function in your servant class. The `clone()` function will be called automatically whenever the threading model demands a new servant instance. Example 32 shows the default implementation of the `clone()` function for the servant class, *PortType*`Impl`.

Example 32: *Default Implementation of the clone() Function*

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

Step 2—Register the wrapper servant

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

For example, Example 33 shows how the main function of the bank server example can be modified to register the `BankImpl` servant with a `PerThread` threading model.

Example 33: *Registering a Servant with a PerThread Threading Model*

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

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

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

Example 33: *Registering a Servant with a PerThread Threading Model*

```
2       bus->register_servant(
            per_thread_bank,
            "../wsdl/bank.wsdl",
            service_name
        );

        IT_Bus::run();

        bus->deregister_servant(service_name);
    }
    catch (IT_Bus::Exception& e) { ... }
```

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

1. In this step, the `BankImpl` servant is wrapped by a new `IT_Bus::PerThreadServant` instance.
2. When it comes to registering, you must register the *wrapper servant*, `per_thread_bank`, instead of the original servant, `my_bank`.

Thread Pool Configuration

Thread pool settings

The thread pool for each port is controlled by the following parameters (which can be set in the configuration):

- *Initial threads*—the number of threads initially created for each port.
- *Low water mark*—the size of the dynamically allocated pool of threads will not fall below this level.
- *High water mark*—the size of the dynamically allocated pool of threads will not rise above this level.

Thread pools are configured by adding to or editing the settings in the *ArtixInstallDir*`/artix/`*Version*`/etc/domains/artix.cfg` configuration file. In the following examples, it is assumed that the Artix application specifies its configuration scope to be `sample_config`.

Note: You can specify the configuration scope at the command line by passing the switch `-ORBname` *ConfigScopeName* to the Artix executable. Command-line arguments are normally passed to `IT_Bus::init()`.

Thread pool configuration levels

Thread pools can be configured at several levels, where the more specific configuration settings take precedence over the less specific, as follows:

- Global level.
- Service name level.
- Qualified service name level.

Global level

The variables shown in Example 34 can be used to configure thread pools at the global level; that is, these settings would apply to all services by default.

Example 34: *Thread Pool Settings at the Global Level*

```
# Artix configuration file

sample_config {
    ...
    # Thread pool settings at global level
    thread_pool:initial_threads = "3";
    thread_pool:low_water_mark  = "5";
    thread_pool:high_water_mark = "10";
};
```

The default settings are as follows:

```
thread_pool:initial_threads = "2";
thread_pool:low_water_mark  = "5";
thread_pool:high_water_mark = "25";
```

Service name level

To configure thread pools at the service name level (that is, overriding the global settings for a specific service only), set the following configuration variables:

`thread_pool:initial_threads:`*ServiceName*
`thread_pool:low_water_mark:`*ServiceName*
`thread_pool:high_water_mark:`*ServiceName*

Where *ServiceName* is the name of the particular service to configure, as it appears in the WSDL `<service name="`*ServiceName*`">` tag.

For example, the settings in Example 35 show how to configure the thread pool for a service named `SessionManager`.

Example 35: *Thread Pool Settings at the Service Name Level*

```
# Artix configuration file

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

Qualified service name level

Occasionally, if the service names from two different namespaces clash, it might be necessary to identify a service by its fully-qualified service name. To configure thread pools at the qualified service name level, set the following configuration variables:

`thread_pool:initial_threads:`*NamespaceURI*`:`*ServiceName*
`thread_pool:low_water_mark:`*NamespaceURI*`:`*ServiceName*
`thread_pool:high_water_mark:`*NamespaceURI*`:`*ServiceName*

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

For example, the settings in Example 36 show how to configure the thread pool for a service named `SessionManager` in the `//my.tns1/` namespace URI.

Example 36: *Thread Pool Settings at the Qualified Service Name Level*

```
# Artix configuration file

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

CHAPTER 3

Artix References

An Artix reference is a handle to a particular service in a particular Bus instance. Because references can be passed around as parameters, they provide a convenient and flexible way of identifying and locating specific services.

In this chapter

This chapter discusses the following topics:

Introduction to References

Overview

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

- A reference is a built-in type in Artix.
- A reference represents a `<wsdl:service>`.
- References can be sent across the wire as parameters of or return values from operations.
- References are fully self-describing. They contain endpoint and contract information in an optimised manner and they can be used either by static or by dynamic clients.
- References are the building blocks for the Artix *Services Locator* and the *Session Manager*, because they allow you to describe Web services that reference other Web services.
- References in Artix are protocol and transport neutral. An Artix reference can be used to represent any WSDL service.

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

Contents of an Artix reference

An Artix reference encapsulates the following data:

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

  Note: If you have loaded the `wsdl_publish` plug-in on the server side, the WSDL location URL will point at a dynamically updated copy of the server's WSDL contract. See page 80.

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

XML representation of a reference

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

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

The schema is also available online at:

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

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

C++ representation of a reference

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

Logical and physical contracts

It is helpful to differentiate between the *logical* and the *physical* parts of a WSDL contract, as follows:

- *Logical contract*—the part of a contract that determines syntax and semantics. In WSDL, a logical contract is effectively a combination of a port type and a binding.
- *Physical contract*—the part of a contract that contains a service's connection details. In WSDL, a physical contract can be identified with a service and its port details.

Static references

A *static reference* is a reference for which both the logical contract and the physical contract appear in the WSDL contract. Hence, static references can only be created for services that are explicitly defined in WSDL.

Figure 7 illustrates the relationship between a static reference and the WSDL contract.

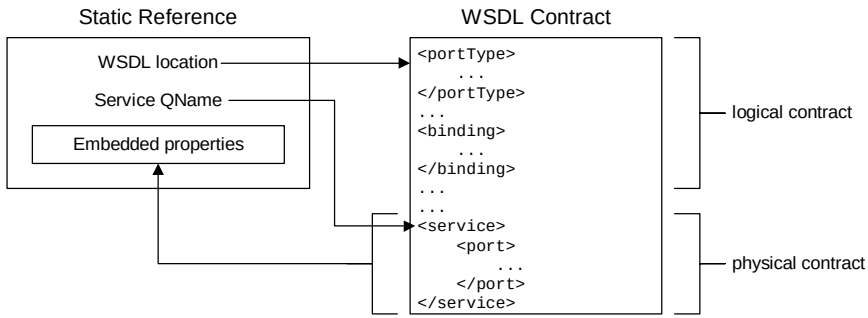

Figure 7: *A Static Reference*

Partial details of the physical contract (from the `<service>` element) are cached in the static reference as *embedded properties*. Only properties that would be relevant to a client are cached in the reference, however.

The version of the physical contract cached in the reference includes dynamically updated data. For example, a port's addressing data would be substituted with the current host name and dynamically allocated IP port.

Transient references

A *transient reference* is a reference for which only the logical contract appears in the WSDL contract. Hence, a transient reference is more flexible, because it can refer to endpoints (represented by a physical contract) created at runtime.

Figure 8 illustrates the relationship between a transient reference and the WSDL contract.

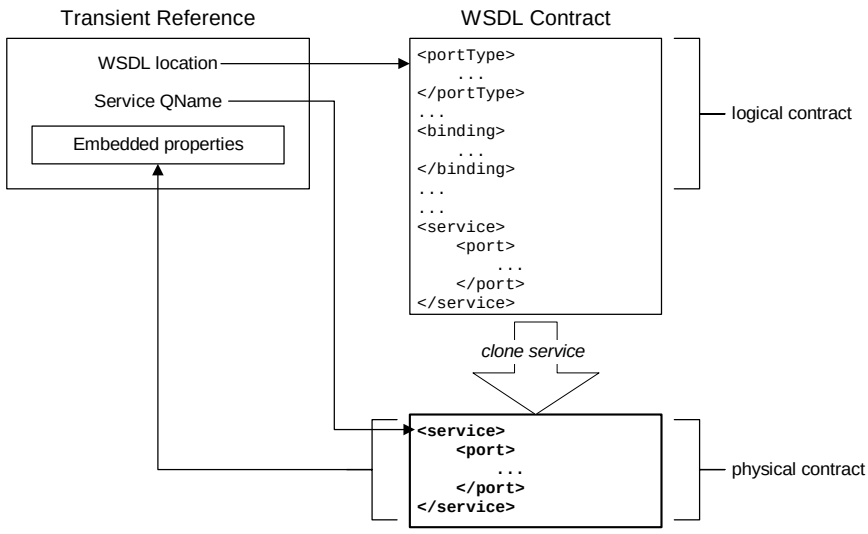

Figure 8: *A Transient Reference*

As shown in Figure 8, the physical contract for a transient reference is created at runtime by cloning the details from an existing `<service>` element. A *cloned service* is created whenever you register a transient servant with the Bus and it has the following characteristics:

- The cloned service is based on an existing `<service>` element that appears in the WSDL contract.
- The clone's service QName is replaced by a dynamically generated, unique service QName.
- The clone's addressing information is replaced such that each address is unique per-clone and per-port.

The WSDL Publish Plug-In

Overview

It is strongly recommended that you activate the *WSDL publish plug-in* for any applications that generate and export Artix references. This is because references are generated with a WSDL location attribute, whose value is virtually unusable unless the WSDL publish plug-in is enabled.

By default, a reference's WSDL location attribute would reference a local file on the server system. This suffers from the following drawbacks:

- It is typically impossible for clients to access the server's copy of the WSDL contract file.
- Endpoint information (the physical contract) might be incomplete, because the server updates transport properties at runtime.

In both of these cases, the client needs to have a way of obtaining the dynamically-updated WSDL contract directly from the remote server. The simplest to achieve this is to configure the server to load the WSDL publish plug-in. The WSDL publish plug-in automatically opens a HTTP port, from which clients can download a copy of the server's in-memory WSDL model.

Loading the WSDL publish plug-in

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

```
# Artix Configuration File
demos{
    server
    {
        orb_plugins = ["xmlfile_log_stream", "wsdl_publish"];
        ...
    };
};
```

Generating references without the WSDL publish plug-in

Figure 9 gives an overview of how an Artix reference is generated when the WSDL publish plug-in is *not* loaded.

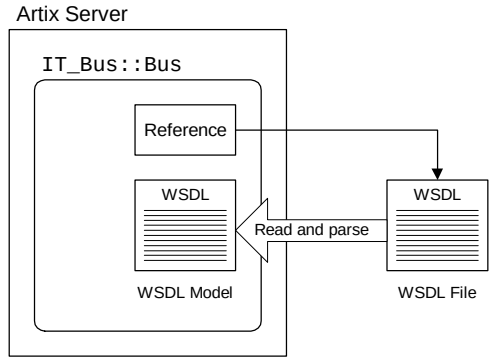

Figure 9: *Generating References without the WSDL Publish Plug-In*

In this case, references generated by the `IT_Bus::Bus` object would, by default, have their WSDL location set to point at the local WSDL file.

This way of setting the WSDL location suffers from the following disadvantages:

- Clients are not able to access the server's WSDL file, unless they happen to share the same file system.
- Even if the server's WSDL file is accessible to a client through a Network File System (NFS) or similar, the WSDL contract in the file does not reflect dynamic updates made by the server (in particular, dynamically allocated IP ports would not be updated in the WSDL file).

WSDL model

When an Artix server starts up, it reads the WSDL files needed by the registered services—for example, in Figure 9, a single WSDL file is read and parsed. After parsing, the WSDL definitions exist in memory in the form of a *WSDL model*. The WSDL model is an XML parse tree containing all the WSDL definitions imported into a particular `IT_Bus::Bus` instance at runtime. Different `IT_Bus::Bus` instances have distinct WSDL models.

API for the WSDL Model

To access the nodes of the WSDL model, you can use the classes defined in the `IT_WSDL` namespace—for example, see the header files in the `include/it_wsdl` directory.

Dynamic Updates

The WSDL model is dynamically updated by the Artix server to reflect changes in the physical contract at runtime. For example, if the server dynamically allocates an IP port for a particular port on a WSDL service, the port's addressing information is updated in the WSDL model.

Generating references with the WSDL publish plug-in

When the WSDL publish plug-in is loaded, the Artix server opens a HTTP port which it uses to publish the in-memory WSDL model. Figure 10 gives an overview of how an Artix reference is generated when the WSDL publish plug-in is loaded.

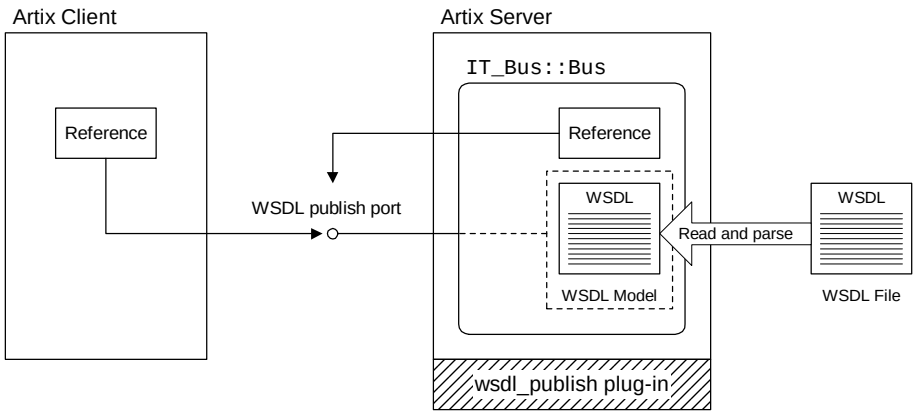

Figure 10: *Generating References with the WSDL Publish Plug-In*

In this case, references generated by the `IT_Bus::Bus` object have their WSDL location set to the following URL:

`http://`*host_name*`:`*WSDL_publish_port*`/`*WSDL_ID*

Where *host_name* is the server host, *WSDL_publish_port* is an IP port used specifically for the purpose of serving up WSDL contracts, and *WSDL_ID* is a proprietary ID that identifies a particular WSDL contract.

If a client accesses the WSDL location URL, the server will convert the WSDL model to XML on the fly and return the resulting WSDL contract in a HTTP message.

Usefulness of the published WSDL model

In most cases, clients do not need to download the published WSDL model at all. Published WSDL is primarily useful for *dynamic clients* that try to invoke an operation on the fly. Such dynamic clients would typically *not* be compiled with Artix stub code. Hence, the only way the clients could obtain the logical contract would be by downloading the published WSDL model.

The published WSDL model can be used as follows, depending on the type of reference:

- *Static reference*—clients can use both the logical contract and the physical contract from the published WSDL model. The physical model for static references is always up-to-date, because of the dynamic updates.
- *Transient reference*—clients can use the logical contract, but *not* the physical contract, from the published WSDL model. Details of the physical contract (actually a cloned service) are available only from the reference's embedded properties.

Multiple Bus instances

Occasionally, you might need to create an Artix server with more than one `IT_Bus::Bus` instance. In this case, you should be aware that separate WSDL models are created for each Bus instance and separate HTTP ports are also opened to publish the WSDL models—see Figure 11.

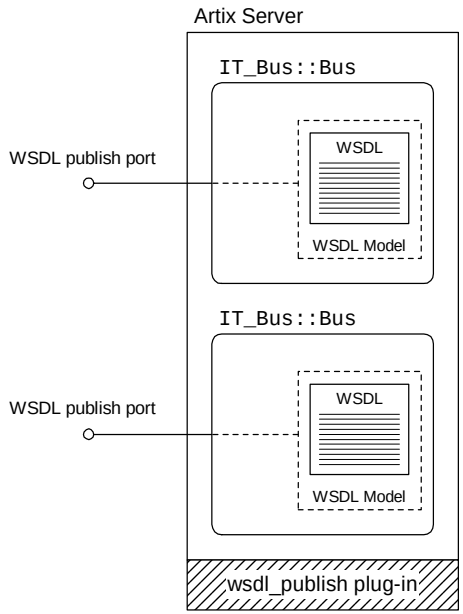

Figure 11: *WSDL Publish Plug-In and Multiple Bus Instances*

Programming with References

Overview

This section explains how to program with Artix references, using a simple bank application as a source of examples. The bank server supports a `create_account()` operation and a `get_account()` operation, which return references to `Account` objects.

To program with references, you need to know how to generate references on the server side and how to resolve references on the client side.

In this section

This section contains the following subsections:

Bank WSDL Contract

Overview

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

The XML Reference type

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

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

The references XML schema

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

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

The schema is also available online at:

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

The Bank example

Figure 12 shows an overview of the Bank example, illustrating how the Bank service uses references to give a client access to a specific account.

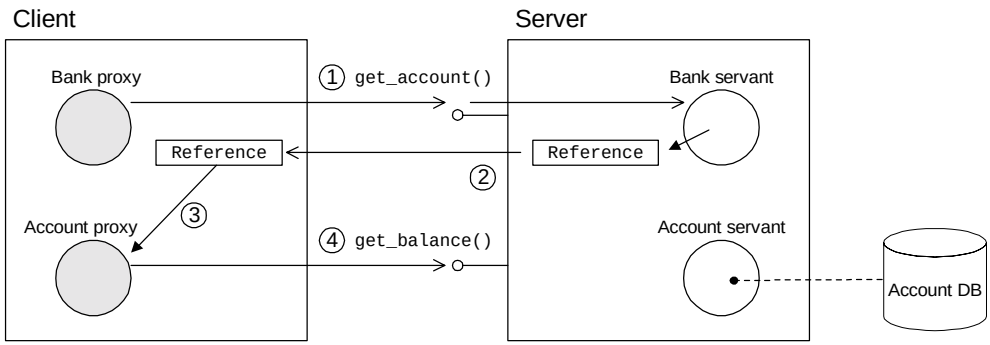

Figure 12: *Using Bank to Obtain a Reference to an Account*

The preceding Bank example can be explained as follows:

1. The client calls `get_account()` on the `BankService` service to obtain a reference to a particular account, *AccName*.
2. The `BankService` creates a reference to the *AccName* account and returns this reference in the response to `get_account()`.
3. The client uses the returned reference to initialize an `AccountClient` proxy.
4. The client invokes operations on the `Account` service through the `AccountClient` proxy.

The Bank WSDL contract

Example 37 shows the WSDL contract for the Bank example that is described in this section. There are two port types in this contract, `Bank` and `Account`. For each of the two port types there is a SOAP binding, `BankBinding` and `AccountBinding`.

Example 37: *Bank WSDL Contract*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:tns="http://www.iona.com/bus/demos/bank"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd1="http://soapinterop.org/xsd"
        xmlns:stub="http://schemas.iona.com/transports/stub"
        xmlns:http="http://schemas.iona.com/transports/http"
xmlns:http-conf="http://schemas.iona.com/transports/http/configu
   ration"
        xmlns:fixed="http://schemas.iona.com/bindings/fixed"
xmlns:iiop="http://schemas.iona.com/transports/iiop_tunnel"
        xmlns:corba="http://schemas.iona.com/bindings/corba"

   xmlns:ns1="http://www.iona.com/corba/typemap/BasePortType.idl
   "
        xmlns:references="http://schemas.iona.com/references"
        xmlns:mq="http://schemas.iona.com/transports/mq"
        xmlns:routing="http://schemas.iona.com/routing"
        xmlns:msg="http://schemas.iona.com/port/messaging"
        xmlns:bank="http://www.iona.com/bus/demos/bank"
        targetNamespace="http://www.iona.com/bus/demos/bank"
        name="BaseService" >
    <types>
```

Example 37: *Bank WSDL Contract*

```
2       <xsd:import
    schemaLocation="../../../../../schemas/references.xsd"
    namespace="http://schemas.iona.com/references"/>
        <schema elementFormDefault="qualified"
            targetNamespace="http://www.iona.com/bus/demos/bank"
            xmlns="http://www.w3.org/2001/XMLSchema">
            <complexType name="AccountNames">
                <sequence>
                    <element maxOccurs="unbounded" minOccurs="0"
                             name="name" type="xsd:string"/>
                </sequence>
            </complexType>
        </schema>
    </types>

    <message name="list_accounts" />
    <message name="list_accountsResponse">
        <part name="return" type="bank:AccountNames"/>
    </message>

    <message name="create_account">
        <part name="account_name" type="xsd:string"/>
    </message>
    <message name="create_accountResponse">
3       <part name="return" type="references:Reference"/>
    </message>

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

    <message name="delete_account">
        <part name="account_name" type="xsd:string"/>
    </message>
    <message name="delete_accountResponse" />

    <message name="get_balance"/>
    <message name="get_balanceResponse">
        <part name="balance" type="xsd:float"/>
    </message>

    <message name="deposit">
```

Example 37: *Bank WSDL Contract*

```
        <part name="addition" type="xsd:float"/>
    </message>

    <message name="depositResponse"/>


    <portType name="Bank">
        <operation name="list_accounts">
            <input name="list_accounts"
                   message="tns:create_account"/>
            <output name="list_accountsResponse"
                    message="tns:list_accountsResponse"/>
        </operation>

5       <operation name="create_account">
            <input name="create_account"
                   message="tns:create_account"/>
            <output name="create_accountResponse"
                    message="tns:create_accountResponse"/>
        </operation>

6       <operation name="get_account">
           <input name="get_account" message="tns:get_account"/>
            <output name="get_accountResponse"
    message="tns:get_accountResponse"/>
        </operation>

        <operation name="delete_account">
            <input name="delete_account"
                   message="tns:delete_account"/>
            <output name="delete_accountResponse"
                    message="tns:delete_accountResponse"/>
        </operation>
    </portType>

    <portType name="Account">
        <operation name="get_balance">
           <input name="get_balance" message="tns:get_balance"/>
            <output name="get_balanceResponse"
                    message="tns:get_balanceResponse"/>
        </operation>
        <operation name="deposit">
            <input name="deposit" message="tns:deposit"/>
            <output name="depositResponse"
                    message="tns:depositResponse"/>
```

Example 37: *Bank WSDL Contract*

```
        </operation>
    </portType>

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

Example 37: *Bank WSDL Contract*

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

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

Example 37: *Bank WSDL Contract*

```
            </input>
            <output>
                <soap:body use="literal"
   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   namespace="http://www.iona.com/bus/demos/bank"/>
            </output>
        </operation>
    </binding>
7   <service name="BankService">
        <port name="BankPort" binding="tns:BankBinding">
            <soap:address
           location="http://localhost:0/BankService/BankPort/"/>
        </port>
    </service>
    <service name="BankServiceRouter">
        <port name="BankPort" binding="tns:BankBinding">
            <soap:address

   location="http://localhost:0/BankService/BankPort/"/>
        </port>
    </service>
8   <service name="AccountService">
        <port name="AccountPort" binding="tns:AccountBinding">
            <soap:address location="http://localhost:0" />
        </port>
    </service>
</definitions>
```

The preceding WSDL contract can be described as follows:

1. The `<definitions>` tag associates the `references` prefix with the `http://schemas.iona.com/references` namespace URI. This means that the reference type is identified as `references:Reference`.
2. The `xsd:import` imports the `<references:Reference>` type definition from the references schema, `references.xsd`. You must edit this line if the references schema is stored at a different location relative to the bank WSDL file.

> **Note:** Alternatively, you could cut and paste the references schema directly into the WSDL contract at this point, replacing the `xsd:import` element.

3. The `create_accountResponse` message (which is the `out` parameter of the `create_account` operation) is defined to be of `references:Reference` type.
4. The `get_accountResponse` message (which is the `out` parameter of the `get_account` operation) is defined to be of `references:Reference` type.
5. The `create_account` operation defined on the `Bank` port type is defined to return a `references:Reference` type.
6. The `get_account` operation defined on the `Bank` port type is defined to return a `references:Reference` type.
7. The information contained in this `<service name="BankService">` element is approximately the same as the information that is held in a `BankService` static reference, apart from the addressing information in the `<soap:address>` element.

   The `BankService` static reference generated at runtime would replace the `http://localhost:0/BankService/BankPort/` SOAP address with `http://`*`host_name`*`:`*`IP_port`*`/BankService/BankPort/` where *host_name* and *IP_port* are substituted with the port address that the server is actually listening on (dynamic port allocation).

   Note: If the IP port in the WSDL contract is non-zero, Artix uses the specified port instead of performing dynamic port allocation. The hostname would still be substituted, however.

8. The information contained in this `<service name="AccountService">` element serves as a prototype for generating `AccountService` transient references.

   An `AccountService` transient reference is cloned from the `AccountService` service at runtime by altering the following data:

   - The service QName is replaced by a transient service QName, which consists of `AccountService` concatenated with a unique ID code.

- The `http://localhost:0` SOAP address is replaced by `http://`*host_name*`:`*IP_port*`/`*TransientURLSuffix*, where *host_name* and *IP_port* are set to the port address that the server is listening on and *TransientURLSuffix* is a suffix that is unique for each transient reference.

Creating References

Overview

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

The following topics are discussed in this section:

- Factory pattern.
- Creating a static reference.
- Creating a transient reference.

Factory pattern

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

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

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

Creating a static reference

Example 38 shows how to create a static `BankService` reference. The distinguishing feature of a static reference is that it is generated from a static servant object.

Example 38: *Creating a Static Reference*

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

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

1   BankImpl my_bank(bus);
```

Example 38: *Creating a Static Reference*

```
2       IT_Bus::Service & service = bus->register_servant(
            my_bank,
            "../wsdl/bank.wsdl",
            service_name,
            "BankPort"
        );

3       IT_Bus::Reference& bank_reference = service->get_reference();
        ...
}
```

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

1. This line creates a `BankImpl` servant instance, which implements the `Bank` port type.
2. The `register_servant()` function registers a static servant instance, taking the following arguments:
   - Servant instance.
   - WSDL file location.
   - Service QName.
   - Port name (optional).

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

   The return value is an `IT_Bus::Service` object, which references the original `BankService` WSDL service.
3. The `get_reference()` function returns an Artix reference for the service object, `service`.

Creating a transient reference

Example 39 gives the implementation of the `BankImpl::create_account()`, function which shows how to create a transient `AccountService` reference. The distinguishing feature of a transient reference is that it is generated from a transient servant object.

Example 39: *Creating a Transient Reference*

```
// C++
void
BankImpl::create_account(
    const IT_Bus::String &account_name,
    IT_Bus::Reference &account_reference
) IT_THROW_DECL((IT_Bus::Exception))
{
    AccountMap::iterator account_iter = m_account_map.find(
                                            account_name
                                        );
    if (account_iter == m_account_map.end())
    {
        cout << "Creating new account: "
             << account_name.c_str() << endl;

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

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

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

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

Example 39: *Creating a Transient Reference*

```
}
```

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

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

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

   The return value is an `IT_Bus::Service` object, which references a WSDL service cloned from `AccountService`.
3. The `get_reference()` function returns an Artix reference for the account service object.

Resolving References

Overview

To a client, an `IT_Bus::Reference` object is just an opaque token that can be used to open a connection to a particular Artix service. The basic usage pattern on the client side, therefore, is for the client to obtain a reference from somewhere and then uses the reference to initialize a proxy object.

Initializing a client proxy with a reference

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

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

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

Client example

Example 40 shows some sample code from a client that obtains a reference to an `Account` and then uses this reference to initialize an `AccountClient` proxy object.

Example 40: *Client Using an Account Reference*

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

// 1. Retrieve an account reference from the remote Bank object.
IT_Bus::Reference account_reference;
bankclient.get_account("A. N. Other", account_reference);

// 2. Resolve the account reference.
AccountClient account(account_reference);

IT_Bus::Float balance;
account.get_balance(balance);
```

Callbacks

Overview

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

In this section

This section contains the following subsections:

Overview of Artix Callbacks

Overview

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

ArtixInstallDir`/artix/`*Version*`/demos/advanced/callback`

Callbacks rely, essentially, on Artix references. Using references, the client can encapsulate the details of its callback service and pass on these details to the server in a reference parameter. Figure 13 illustrates how this process works.

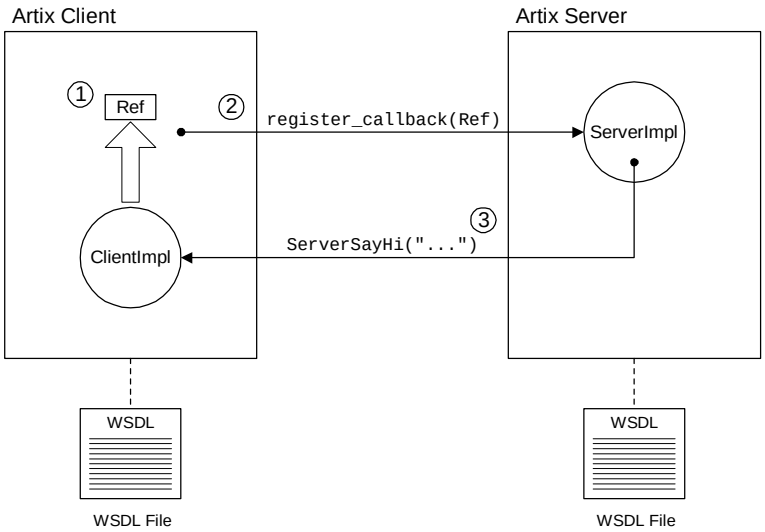

Figure 13: *Overview of the Callback Demonstration*

Callback steps

Example 13 on page 101 shows the callback proceeding according to the following steps:

1. After the basic initialization steps, including registration of the `ClientImpl` servant and `ClientService` service, the client generates a reference for the callback service.

   The client callback service is activated and capable of receiving incoming invocations as soon as it is registered.
2. The client calls `register_callback()` on the remote server, passing the reference generated in the previous step.
3. When the server receives the callback reference, it immediately calls back on the `ClientImpl` servant by invoking `ServerSayHi()`.

> **Note:** In a more realistic application, it is likely that the server would cache a copy of the callback reference and call back on the client at a later time, instead of calling back immediately.

Threading

By default, both the client and the server allocate a pool of threads to process incoming requests (see "Multi-Threading" on page 62). One of the positive side effects of this policy is that the callback scenario shown in Figure 13 on page 101 is *not* subject to deadlock.

> **Note:** In the current example, it is also significant that the client service is activated as soon as it is registered. Otherwise the code shown in Example 42 on page 109 would lead to deadlock.

Routing and Callbacks

Overview

Callbacks are fully compatible with Artix routers. Reference that passes through a router are automatically *proxified*, if necessary. Proxification means that the router automatically creates a new route for the references that pass through it.

Note: Proxification is not necessary, if the transport protocols along the route are the same.

For example, consider the callback routing scenario shown in Figure 14. In this scenario, a SOAP/HTTP Artix server replaces a legacy CORBA server. As part of a migration strategy, legacy CORBA clients can continue to communicate with the new server by interposing an Artix router to translate between the IIOP and SOAP/HTTP protocols.

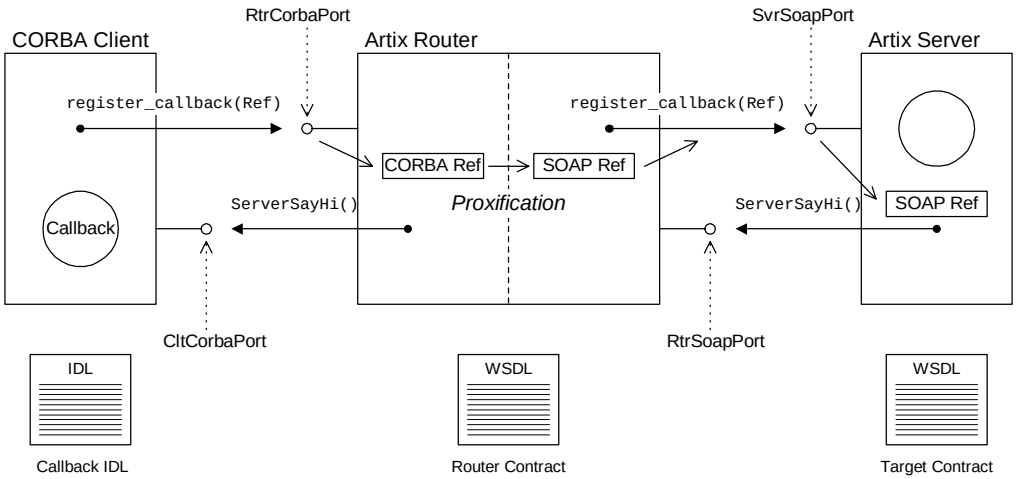

Figure 14: *Overview of a Callback Routing Scenario*

Contracts

The applications in Figure 14 are associated with three distinct, but related, contracts as follows:

- Callback IDL.
- Target contract.
- Router contract.

Callback IDL

The CORBA client uses a contract coded in OMG Interface Definition Language (IDL). This IDL contract defines both the target interface (implemented by the Artix server) and the callback interface (implemented by the CORBA client).

Target contract

In this scenario, the target contract is generated from the callback IDL using the IDL-to-WSDL compiler. Hence, this WSDL contract contains both the target interface and the callback interface as WSDL port types.

The target contract also contains a single WSDL service description, which includes the `SvrSoapPort` port.

Router contract

The router contract holds details about the CORBA side of the application as well as the SOAP/HTTP side, including the following information:

- Target WSDL port type.
- Callback WSDL port type.
- CORBA WSDL binding for the target.
- SOAP/HTTP WSDL binding for the target.
- CORBA WSDL service, containing the `RtrCorbaPort` port.
- SOAP/HTTP WSDL service, containing the `SvrSoapPort` port.
- Prototype SOAP/HTTP WSDL service, needed for generating the transient endpoint with `RtrSoapPort` port.
- Route information.

You can generate a router contract using the Artix Designer GUI tool. To specify the location of the generated router contract, you can set the `plugins:routing:wsdl_url` configuration variable in the router scope of the `artix.cfg` configuration file.

Routes

As shown in Figure 14 on page 103, the following routes are created in this scenario:

- *Client-Router-Target route*—this route is documented explicitly in the router contract. The source port, `RtrCorbaPort`, and the destination port, `SvrSoapPort`, are described in the router contract.

  For example, when the client calls the `register_callback()` operation, the request travels initially to the `RtrCorbaPort` on the router (over IIOP) and then on to the `SvrSoapPort` on the target server (over SOAP/HTTP).
- *Target-Router-Client route (callback route)*—the reverse route (for callbacks) is *not* documented explicitly in the router contract. This route is constructed at runtime to facilitate routing callback invocations.

  For example, when the Artix server calls the `ServerSayHi()` callback operation, the request travels to the `RtrSoapPort` on the router (over SOAP/HTTP) and then on to the `CltCorbaPort` on the client (over IIOP).

Proxification

Proxification refers to the process whereby a reference of a certain type (for example, a CORBA reference) that passes through the router is automatically converted to a reference of another type (for example, an Artix SOAP reference).

The proxification process is of key importance to Artix callbacks. If the router in Figure 14 on page 103 did not proxify `register_callback()`'s reference argument, it would be impossible for the server to call back on the client. The server can communicate *only* with SOAP/HTTP endpoints, not with IIOP endpoints.

In Figure 14 on page 103, the router proxifies the callback reference as follows:

1. When the `register_callback()` operation is invoked, the router recognizes that the reference argument must be converted into a SOAP/HTTP-format reference.
2. The router dynamically creates a new service and port, `RtrSoapPort`, to receive callback requests in SOAP/HTTP format. The new service is a transient service cloned from a service in the router WSDL contract. The router looks for a service that satisfies the following criteria:
   - Supports the same port type as the original reference.
   - Supports the same type of binding (for example, SOAP or CORBA) as the target server.
3. The router creates a new SOAP/HTTP reference, encapsulating details of the `RtrSoapPort` endpoint.
4. The router forwards the `register_callback()` operation on to the target server in SOAP format, with the proxified SOAP/HTTP reference as its argument.
5. The router dynamically constructs a callback route, with source port, `RtrSoapPort`, and destination port, `CltCorbaPort`.

Callback WSDL Contract

Overview

This subsection describes the WSDL contract that defines the interaction between the client and the server in the callback demonstration. This WSDL contract is somewhat unusual in that it defines port types both for the client and for the server applications.

WSDL contract

Example 41 shows the WSDL contract used for the callback demonstration.

Example 41: *Example Callback WSDL Contract*

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

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

Example 41: *Example Callback WSDL Contract*

```
        </operation>
    </portType>
    <portType name="ServerPortType">
        <operation name="register_callback">
            <input message="tns:register_callback"
                   name="register_callback"/>
        </operation>
    </portType>
    ...
    <service name="ClientService">
        ...
    </service>
    <service name="ServerService">
        ...
    </service>
</definitions>
```

Port types and operations

The WSDL contract in Example 41 on page 107 defines the following port types and operations:

- `ServerPortType` port type—implemented on the server side. This server port type supports a single WSDL operation:
  - `register_callback` operation—takes a single Artix reference argument, which is used to pass a reference to the client callback object.
- `ClientPortType` port type—implemented on the client side. This callback port type supports a single WSDL operation:
  - `ServerSayHi` operation—takes a single string argument. The server calls back on this operation after it has received a reference to the client's service.

Client Implementation

Overview

In a callback scenario, the client plays a hybrid role: part client, part server. Hence, the implementation of the callback client includes coding steps you would normally associate with a server, including an implementation of a servant class. The callback client implementation consists of two main parts, as follows:

- Client main function.
- ClientImpl servant class.

Client main function

Example 42 shows the code for the callback client main function, which instantiates and registers a `ClientImpl` servant before calling on the remote server to register the callback.

Example 42: *Callback Client Main Function*

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

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

IT_USING_NAMESPACE_STD

using namespace DemosCallback;
using namespace IT_Bus;

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

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

1       ClientImpl servant(bus);
        cout << "Activating Service on Bus" << endl;
```

Example 42: *Callback Client Main Function*

```
        QName service_qname(
            "", "ClientService", "http://www.iona.com/callback"
        );
        Service & service =
            bus->register_servant(
                servant,
                "../../etc/callback.wsdl",
                service_qname
            );

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

        cout << "Callback Service Ready." << endl;
        bus->run();

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

The preceding code example can be explained as follows:

1. The `ClientImpl` servant class implements the `ClientPortType` port type. The `ClientImpl` instance created on this line is the client callback object.
2. The `service_qname` specifies the WSDL service to be activated on the client side. This QName refers to the `<service name="ClientService">` element in Example 41 on page 107.
3. Register the callback servant with the Bus, thereby activating the `ClientService` service. From this point on, the `ClientService` service is active and able to process incoming callback requests in a background thread.

4. A reference to the callback service is generated by calling `IT_Bus::Service::get_reference()`.
5. This line invokes the `register_callback()` operation on the remote server, passing in the reference to the client callback object. From this point on, the server could invoke an operation on the callback.
6. Just as in a normal server, the callback client calls the blocking `IT_Bus::run()` function to allow the application to process incoming requests.

ClientImpl servant class

Example 43 shows the implementation of the `ClientImpl` servant class, which is responsible for receiving the `ClientImpl::ServerSayHi()` callback from the server. The implementation of this servant class is trivial. It follows the usual pattern for a servant class implementation and the `ServerSayHi()` function simply prints out its string argument.

Example 43: *ClientImpl Servant Class Implementation*

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

IT_USING_NAMESPACE_STD
using namespace DemosCallback;

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

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

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

Example 43: *ClientImpl Servant Class Implementation*

```
    cout <<"ClientImpl::ServerSayHi() ended"<<endl;
}
```

Server Implementation

Overview

The implementation of the server in this callback example follows the usual pattern for an Artix server. The server main function instantiates and registers a servant object. A separate file contains the implementation of the servant class, `ServerImpl`. The server implementation thus consists of two main parts, as follows:

- Server main function.
- ServerPortType implementation.

Server main function

Example 44 shows the code for the server main function, which instantiates and registers a `ServerImpl` servant. The server then waits for the client to register a callback using the `register_callback` operation.

Example 44: *Server Main Function*

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

#include "ServerImpl.h"

IT_USING_NAMESPACE_STD

using namespace IT_Bus;
using namespace DemosCallback;

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

1       ServerImpl servant(bus);
2       IT_Bus::QName service_qname(
            "", "ServerService", "http://www.iona.com/callback"
```

Example 44: *Server Main Function*

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

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

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

The preceding code example can be explained as follows:

1. The `ServerImpl` servant class implements the `ServerPortType` port type, which supports the `register_callback` operation.
2. The `service_qname` refers to the `<service name="ServerService">` element in Example 41 on page 107.
3. Register the `ServerImpl` servant with the Bus, thereby activating the `ServerService` service.
4. Call the blocking `IT_Bus::run()` function to allow the server application to process incoming requests.

ServerPortType implementation

Example 45 shows the implementation of the `ServerImpl` servant class. There is just one WSDL operation, `register_callback()`, to implement in this class.

Example 45: *ServerImpl Servant Class Implementation*

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

Example 45: *ServerImpl Servant Class Implementation*

```
IT_USING_NAMESPACE_STD
using namespace DemosCallback;

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

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

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

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

The preceding code example can be explained as follows:

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

   This example, where the callback is invoked within the body of `register_callback()`, is a little bit artificial. In a more typical use case, the server would cache an instance of the callback client proxy and then call back later, in response to some event that is of interest to the client.

CHAPTER 4

The Artix Locator

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

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

In this chapter

This chapter discusses the following topics:

Overview of the Locator

Overview

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

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

Figure 15 gives a general overview of the locator architecture.

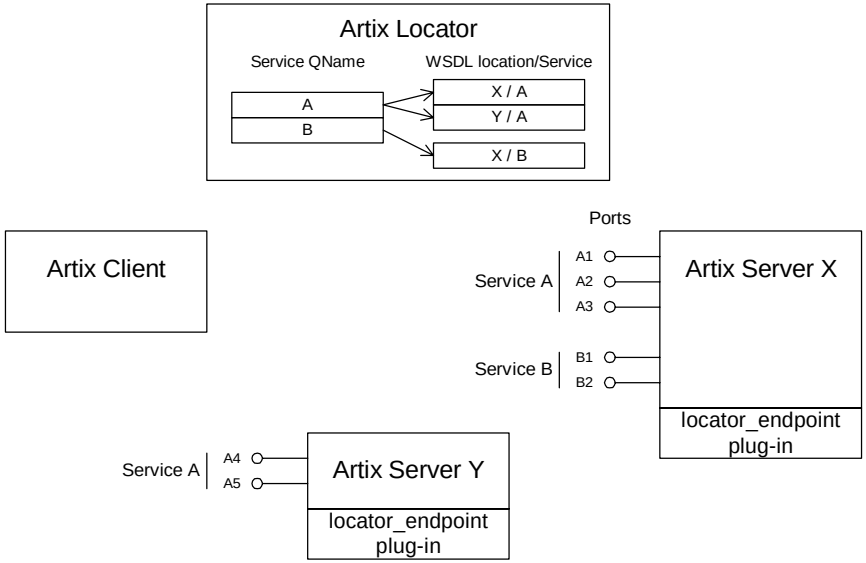

Figure 15: *Artix Locator Overview*

Locator demonstration

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

ArtixInstallDir`/artix/`*Version*`/demos/uncategorized/locator`

Locator service

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

- *Standalone deployment*—the locator is deployed as an independent server process (as shown in Figure 15). This approach is described in detail in the "Using the Artix Locator Service" chapter from the *Artix User's Guide*. Sample source code for such a standalone locator service is provided in the `demos/uncategorized/locator` demonstration.
- *Embedded deployment*—the locator is deployed by embedding it within another Artix server process. This approach is possible because the locator is implemented as a plug-in, which can be loaded into any Artix application.

Endpoint definition

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

Registering endpoints

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

Looking up references

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

Load balancing with the locator

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

For example, Figure 15 on page 118 shows the `Service A` QName with two endpoints registered against it:

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

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

Locator WSDL

Overview

The locator WSDL contract, `locator.wsdl`, defines the public interface of the locator through which the service can be accessed either locally or remotely. This section shows extracts from the locator WSDL that are relevant to normal user applications. The following aspects of the locator WSDL are described here:

- Binding and protocol.
- WSDL contract.
- C++ mapping.

Binding and protocol

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

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

WSDL contract

Example 46 shows an extract from the locator WSDL contract that focuses on the aspects of the contract relevant to an Artix application programmer. There is just one WSDL operation, `lookup_endpoint`, that an Artix client typically needs to call.

Example 46: *Extract from the Locator WSDL Contract*

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:ref="http://schemas.iona.com/references"
    xmlns:ls="http://ws.iona.com/locator"
    targetNamespace="http://ws.iona.com/locator">
     <types>
         <xs:schema targetNamespace="http://ws.iona.com/locator">
1            <xs:import
    schemaLocation="../../../schemas/references.xsd"
    namespace="http://schemas.iona.com/references"/>
```

Example 46: *Extract from the Locator WSDL Contract*

```
            ...
2           <xs:element name="lookupEndpoint">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="service_qname"
                                    type="xs:QName"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
3           <xs:element name="lookupEndpointResponse">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="service_endpoint"
                                    type="ref:Reference"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:complexType
    name="EndpointNotExistFaultException">
                <xs:sequence>
                    <xs:element name="error" type="xs:string"/>
                </xs:sequence>
            </xs:complexType>
4           <xs:element name="EndpointNotExistFault"
                        type="ls:EndpointNotExistFaultException"/>
        </xs:schema>
    </types>
    ...
    <message name="lookupEndpointInput">
        <part name="parameters" element="ls:lookupEndpoint"/>
    </message>
    <message name="lookupEndpointOutput">
        <part name="parameters"
    element="ls:lookupEndpointResponse"/>
    </message>
    <message name="endpointNotExistFault">
        <part name="parameters"
    element="ls:EndpointNotExistFault"/>
    </message>

5   <portType name="LocatorService">
        ...
6       <operation name="lookup_endpoint">
            <input message="ls:lookupEndpointInput"/>
            <output message="ls:lookupEndpointOutput"/>
```

Example 46: *Extract from the Locator WSDL Contract*

```
            <fault name="fault"
                   message="ls:endpointNotExistFault"/>
        </operation>
    </portType>
    <binding name="LocatorServiceBinding"
             type="ls:LocatorService">
        ...
    </binding>
    <service name="LocatorService">
        <port name="LocatorServicePort"
              binding="ls:LocatorServiceBinding">
            <soap:address
 location="http://localhost:0/services/locator/LocatorService"/>
        </port>
    </service>
</definitions>
```

The preceding locator WSDL extract can be explained as follows:

1. This line imports the schema definition of the `ref:Reference` type. You might have to edit the value of the `schemaLocation` attribute, if the `references.xsd` schema file is stored in a different location relative to the `locator.wsdl` file.
2. The `lookupEndpoint` type is the input parameter type for the `lookup_endpoint` operation. It contains just the QName (qualified name) of a particular WSDL service.
3. The `lookupEndpointResponse` type is the output parameter type for the `lookup_endpoint` operation. It contains an Artix reference for the specified service. If more than one endpoint is registered against a particular service name, the locator picks one of the endpoints using a round-robin algorithm.
4. The `EndpointNotExist` fault would be thrown if the `lookup_endpoint` operation fails to find an endpoint registered against the requested service type.
5. The `LocatorService` port type defines the public interface of the Artix locator service.
6. The `lookup_endpoint` operation, which is called by Artix clients to retrieve endpoint references, is the only operation from the `LocatorService` port type that user applications would typically need.

7. The SOAP `location` attribute specifies the host and IP port for the locator service. If you want the locator to run on a different host and listen on a different IP port, you should edit this setting.

C++ mapping

Example 47 shows an extract from the C++ mapping of the `LocatorService` port type. This extract shows only the `lookup_endpoint` WSDL operation—the other WSDL operations in this class are normally not needed by user applications.

Example 47: *C++ Mapping of the LocatorService Port Type*

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

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

      private:

      public:
        LocatorServiceClient(
            IT_Bus::Bus_ptr bus = 0
        );

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

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

        LocatorServiceClient(
```

Example 47: *C++ Mapping of the LocatorService Port Type*

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

        ~LocatorServiceClient();
        ...
        virtual void
        lookup_endpoint(
            const IT_Bus_Services::lookupEndpoint &
                                            lookupEndpoint_in,
            IT_Bus_Services::lookupEndpointResponse &
                                        lookupEndpointResponse_out
        ) IT_THROW_DECL((IT_Bus::Exception));
    };
};
```

The lookupEndpoint type

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

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

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

The lookupEndpointResponse type

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

```
// C++
namespace IT_Bus_Services
{
    class lookupEndpointResponse
      : public IT_Bus::SequenceComplexType
    {
      public:
        lookupEndpointResponse();
        lookupEndpointResponse(const lookupEndpointResponse&
   copy);
        virtual ~lookupEndpointResponse();
        ...
        const IT_Bus::Reference & getservice_endpoint() const;
        IT_Bus::Reference &       getservice_endpoint();
        void setservice_endpoint(const IT_Bus::Reference & val);
        ...
    };
};
```

Registering Endpoints with the Locator

Overview

To register a server's endpoints with the locator, you must configure the server to load a specific set of plug-ins. Once the appropriate plug-ins are loaded, the server will automatically register every endpoint (that is, service/port combination) that is created on the server side.

There is currently no programming API for registering endpoints explicitly.

Configuring a server to register endpoints

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

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

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

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

References

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

- "Using the Artix Locator Service" chapter from the *Artix User's Guide*.
- The Artix `locator` demonstration in `artix/`*Version*`/demos/uncategorized/locator`.

Reading a Reference from the Locator

Overview

After the target server (in this example, the `SimpleService` server) has started up and registered its endpoints with the locator, an Artix client can then bootstrap a connection to the target server by reading one of its endpoint references from the locator. Figure 16 shows an outline of how a client bootstraps a connection in this way.

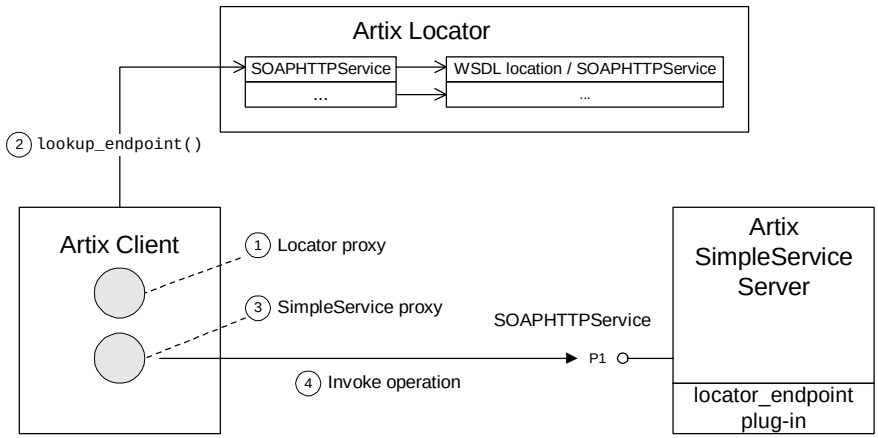

Figure 16: *Steps to Read a Reference from the Locator*

Programming steps

The main programming steps needed to read a reference from the locator, as shown in Figure 16, are as follows:

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

Example

Example 48 shows an example of the code for an Artix client that retrieves a reference to a `SimpleService` service from the Artix locator.

Example 48: *Example of Reading a Reference from the Locator Service*

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

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

IT_USING_NAMESPACE_STD
using namespace IT_Bus;
using namespace IT_Bus_Services;
using namespace SimpleServiceNS;

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

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

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

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

4       String port_name("LocatorServicePort");

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

Example 48: *Example of Reading a Reference from the Locator Service*

```
            );

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

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

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

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

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

1. You should ensure that the client picks up the correct configuration by passing the appropriate value of the `-ORBname` parameter. In this example, the `-ORBname` parameter is hard-coded, but you might prefer to take this parameter from the command line instead.
2. This line constructs a qualified name, `service_name`, that identifies the `<service name="LocatorService">` tag from the locator WSDL. See the listing of the locator WSDL in Example 46 on page 121.
3. This line constructs a qualified name, `sh_service_name`, that identifies the `SOAPHTTPService` service from the `SimpleService` WSDL.
4. This port name refers to the `<port name="LocatorServicePort" ...>` tag in the locator WSDL (see Example 46 on page 121).
5. The locator service proxy is created by calling the three-argument constructor for the `LocatorServiceClient` class. The three arguments passed (locator WSDL, service name, and port name) specify the locator endpoint exactly.
6. The `lookup_endpoint()` operation is invoked on the locator to find an endpoint of `SOAPHTTPService` type (specified in the `sh_input` parameter).

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

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

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

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

Pausing and Resuming Endpoints

Overview

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

IT_Bus::Service pause and resume functions

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

IT_Bus::Service::reached_capacity()

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

IT_Bus::Service::below_capacity()

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

C++ server example

Example 49 shows how to pause and resume the endpoints for a `BookService` service.

Example 49: *Code to Pause and Resume a Service's Endpoints*

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

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

CHAPTER 5

Using Sessions in Artix

The Artix Session Manager helps you manage service resources.

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

In this chapter

This chapter discusses the following topics:

Introduction to Session Management in Artix

Overview

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

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

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

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

How do the plug-ins interact?

Figure 17 shows a diagram of how the session manager plug-ins are deployed in an Artix System. As you can see the session manager service plug-in and the policy callback plug-in are both deployed into the same process. While in this example, they are deployed into a standalone service, they can be deployed in any Artix process. The session manager service plug-in and the policy plug-in interact to ensure that the session manager does not hand out sessions that violate the policies established by the policy plug-in.

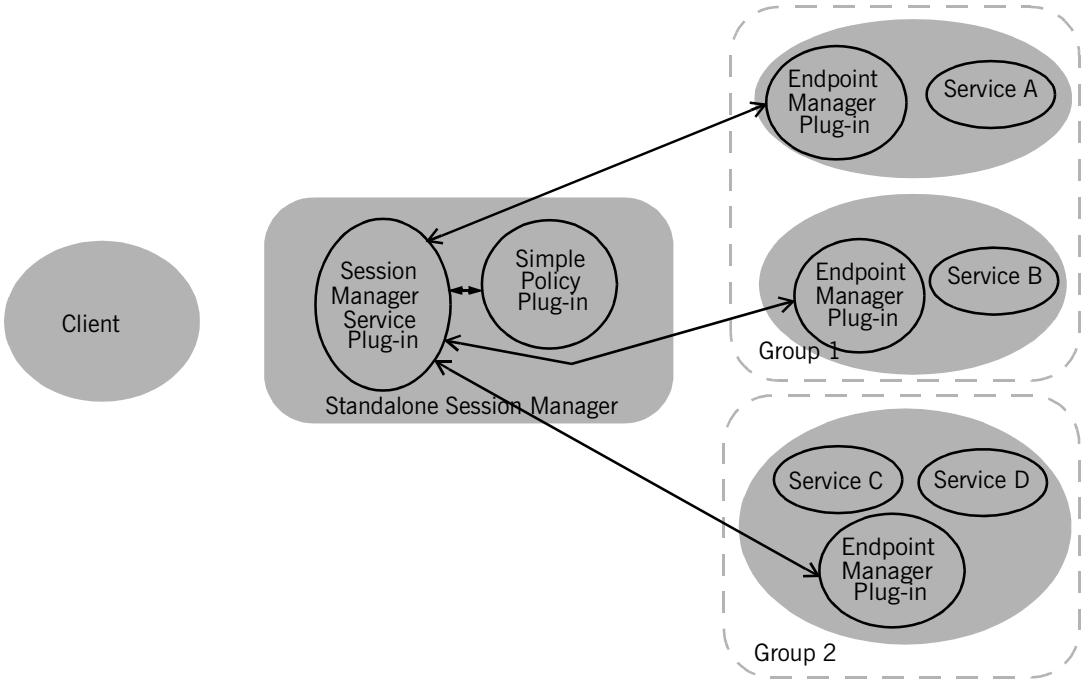

Figure 17: *The Session Manager Plug-ins*

The endpoint manager plug-ins are deployed into the server processes which contain session managed services. A process can host two services, like *Service C* and *Service D* in Figure 17, but the process will have only one endpoint manager. The endpoint manager plug-ins are in constant communication with the session manager service plug-in to report on

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

What are sessions?

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

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

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

What are groups?

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

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

Registering a Server with the Session Manager

Overview

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

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

Configuring the server

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

- `soap`
- `http`
- `session_endpoint_manager`

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

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

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

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

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

Example 50 shows the configuration scope of a server that hosts services managed by the session manager.

Example 50: *Server Configuration Scope*

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

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

For more information on Artix configuration see Deploying and *Managing Artix Solutions*.

You also need to configure the port on which the endpoint manager will run. To do this you modify `session-manager.wsdl`, provided in the `wsdl` folder of your Artix installation, to specify the HTTP address at which the endpoint manager will be available. Using any text editor, open `session-manager.wsdl` and edit the `<soap:address>` entry for the `SessionEndpointManagerService` to specify the proper address. Example 51 shows a modified session manager contract entry. The highlighted part has been modified to point to the desired address.

Example 51: *Endpoint Manager Address*

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

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

Registration

Once a properly configured server starts up, it automatically registers with the session manager specified by the contract pointed to by `plugins:session_endpoint_manager:wsdl_url`.

Working with Sessions

Overview

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

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

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

Instantiating a session manager proxy

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

Example 52 shows how to instantiate a session manager proxy.

Example 52: *Instantiating a Session Manager Proxy*

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

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

Start a session

After instantiating a session manager proxy, a client can then start a session for the desired service's group using the session manager's `begin_session()` method. `begin_session()` has the following signature:

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

`input` contains the name of the desired group and the desired duration of the session. The group name is set using the `setendpoint_group()` method. The group name can be any valid string and corresponds to the default group name set in the service's configuration scope as described in "Configuring the server" on page 139.

The session duration is set using the `setprefered_renew_timeout()` method. The duration is specified in seconds. If the specified duration is less than the value specified by the session manager's `min_session_timeout` configuration setting, it will be set to the configured minimum value. If the specified duration is higher than the value specified by the session manager's `max_session_timeout` configuration setting, it will be set the configured max value.

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

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

Example 53 shows the client code to begin a session for `qajaq_group`.

Example 53: *Beginning a Session*

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

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

session_mgr.begin_session(begin_session_request,
                          begin_session_response);

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

Get a list of endpoints in the group

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

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

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

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

Example 54 shows how to get the list of services for a group.

Example 54: *Retrieving the List of Services in a Group*

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

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

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

Create a proxy for the requested service

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

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

Because the session manager simply returns the services in the order the services registered with the session manager, the clients must be responsible for circulating through the list or else they will all make requests on only one service in the group. Also, because the session manager does not force all members of a group to implement the same interface, you may

want to have your clients check each service to see if it implements the correct interface by checking the reference's service name as shown in Example 55.

Example 55: *Checking the Service Reference for its Interface*

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

Example 56 shows the client code for creating a proxy `qajaq` server from a group service.

Example 56: *Instantiate a Proxy Server*

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

Create a session header

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

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

Setting the proxy to use input message attributes

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

Setting this property to `true` tells the bus to ensure the input message attributes are propagated through to the server. Example 57 shows how to set the client proxy port's `use_input_message_attributes` property to `true`.

Example 57: *Use Input Message Attributes*

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

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

Getting a handle to the input message attributes

A pointer to the proxy port's input message attributes is returned by the port's `get_input_message_attributes()` method. Example 58 shows how to get a handle to the input message attributes.

Example 58: *Getting the Input Message Attributes*

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

Setting the session information into the input message attributes

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

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

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

The input message attributes are set using the message attribute handle's `set_string()` method. `set_string()` takes two attributes. The first is a string specifying the name of the attribute being set. The second is the value to be set for the attribute. Example 59 shows how to set the session information in to the input message attributes.

Example 59: *Setting the Input Message Attributes*

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

Make requests on service proxy

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

Renewing a session

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

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

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

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

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

Example 60 shows how to end a session.

Example 60: *Ending a Session*

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

End the session

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

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

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

Example 61 shows how to end a session.

Example 61: *Ending a Session*

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

CHAPTER 6

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 18.

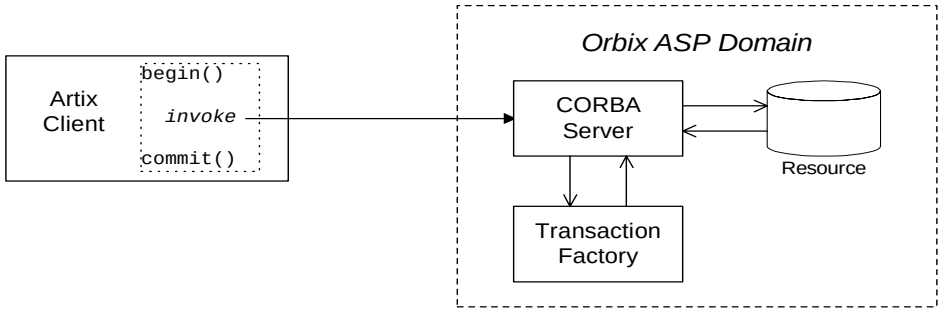

Figure 18: *Artix Client Invokes a Transactional Operation on a CORBA OTS Server*

In Figure 18, 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 62 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 62: *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 63 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 63: *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 7

Artix Contexts

Artix contexts enable you to send additional data with an operation request, without having to declare the data as a parameter. The mechanism for transmitting contexts is binding-specific—for example, the context data might be transmitted in a SOAP header or in a CORBA service context.

In this chapter

This chapter discusses the following topics:

Introduction to Contexts

Overview

This section provides a conceptual overview of Artix contexts, including a brief look at the programming interface required for using contexts with different binding types.

In this section

This section contains the following subsections:

Protocols that Support Contexts

Overview

Artix contexts provide a general purpose mechanism for embedding data in message headers and they are designed to work independently of any particular protocol or binding. Currently, you can embed context data in the following types of protocol header:

- SOAP.
- CORBA.

SOAP

You can send context data in a SOAP header, as shown in Figure 19, by registering a context data type with the `IT_Bus::SoapContextContainer` object.

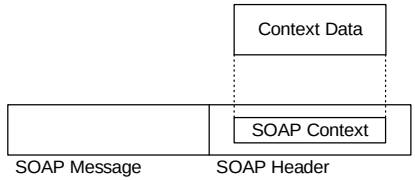

Figure 19: *Inserting Context Data into a SOAP Header*

The context data is sent in an Artix-specific SOAP header, whose format is defined by the `http://schemas.iona.com/custom_header` schema.

CORBA

You can send context data in a CORBA header, as shown in Figure 19, by registering a context data type with the `IT_Bus::CORBAContextContainer` object.

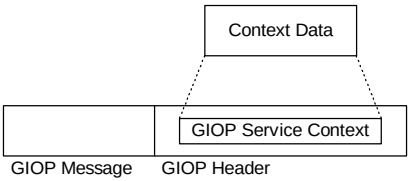

Figure 20: *Inserting Context Data into a GIOP Service Context*

In CORBA, the message formats are defined by the General Inter-ORB Protocol (GIOP) specification. In particular, the GIOP request and reply message formats allow you to include arbitrary header data in *GIOP Service Context*.

Defining Context Data Types

Overview

Although you can use simple data types for context data, in most cases you would want to define a user-defined type, the *context data type*, to represent your context data.

What is a context data type?

A context data type is any schema type derived from `xsd:anyType`. In other words, a context data type can be any simple or complex schema type.

Defining a context schema

It is usually more appropriate to define a context data type (or types) in a separate schema file, rather than including the definition in the application's WSDL contract. This approach is more logical, because contexts are normally used to implement features independently of any particular WSDL contract.

For example, to define a complex context data type, *ContextDataType*, in the namespace, *ContextDataURI*, you could define a context schema following the outline shown in Example 64.

Example 64: *Outline of a Context Schema*

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="ContextDataURI"
    elementFormDefault="qualified"
   attributeFormDefault="unqualified">
    <xs:complexType name="ContextDataType">
        ...
    </xs:complexType>
</xs:schema>
```

Generating stubs for a context schema

To generate C++ stubs from a context schema file, *ContextSchema*`.xsd`, enter the following command at the command line:

`wsdltocpp` *ContextSchema*`.xsd`

The WSDL-to-C++ compiler will generate the following C++ stub files:

ContextSchema`_wsdlTypes.h`
ContextSchema`_wsdlTypesFactory.h`
ContextSchema`_wsdlTypes.cxx`
ContextSchema`_wsdlTypesFactory.cxx`

Registering Context Types

Overview

Figure 21 shows an overview of what happens when you register a context data type with a context container.

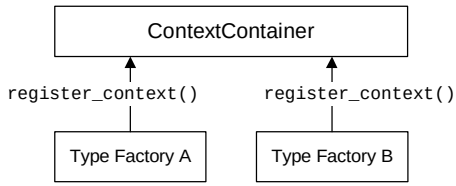

Figure 21: *Registering Context Types with a Context Container*

You register a context type by calling a `register_context()` function on a context container instance. Typically, some variants of the `register_context()` function are provided, but all of the variants include a context data type QName as one of the arguments.

The main effect of registering a context type is that the context container adds a type factory reference to its internal table. This type factory reference enables the context container to create context data instances whenever they are needed.

Note: This pre-supposes that the application is linked with the context schema stub code, which creates static instances of the relevant type factories.

Getting a context container instance

To get a reference to a context container instance, you call the `IT_Bus::get_context_container()` function, shown in Example 65.

Example 65: *The IT_Bus::get_context_container() Function*

```
// C++
namespace IT_Bus {
    class IT_BUS_API Bus : public BusPlugInManager
    {
      public:
```

Example 65: *The IT_Bus::get_context_container() Function*

```
        virtual ContextContainer*
        get_context_container(const String& container_name) = 0;
        ...
    };
};
```

The return type of the `get_context_container()` function depends on the string argument, `container_name`. Table 3 shows the values allowed for the `container_name` argument and the corresponding return types.

Table 3: *String Arguments to the get_context_container() Function*

container_name String	Return Type
`SoapContextContainer`	`IT_Bus::SoapContextContainer`
`CorbaContextContainer`	`IT_Bus::CORBAContextContainer`

Registering a SOAP context

Example 66 shows the signature of the `register_context()` function in the `SoapContextContainer` class, which is used to register a context data type with the SOAP context container.

Example 66: *The register_context() Function for SOAP Contexts*

```
// C++
namespace IT_Bus {
    class IT_SOAP_API SoapContextContainer
        : public virtual ContextContainer
    {
      public:
        virtual void
        register_context(
            const QName&  context_type,
            const QName&  message_name,
            const String& part_name
        ) = 0;
        ...
    };
};
```

The `SoapContextContainer::register_context()` function takes the following arguments:

- `context_type`—the qualified name of the context data type. It can be any schema type (that is, any type derived from `xsd:anyType`).
- `message_name`—the qualified name of the Artix-specific SOAP header type that is used to encapsulates context data in a SOAP header.

  Currently, the only message name you can select is the qualified name with local part, `header_content`, and namespace URI, `http://schemas.iona.com/custom_header`.
- `part_name`—the part of the Artix-specific SOAP header that contains the context data. This argument must have the value, `header_info`.

Registering a CORBA context

Example 67 shows the registration functions in the `CORBAContextContainer` class, which are used to register a context data type with the CORBA context container.

Example 67: *The register_context() Function for CORBA Contexts*

```
// C++
namespace IT_Bus {
    class IT_WS_ORB_API CORBAContextContainer
        : public virtual IT_Bus::ContextContainer
    {
      public:
        virtual void
        register_context(
            const IT_Bus::QName& context_name,
            const IT_Bus::QName& context_type
        ) = 0;

        virtual void
        register_context_as_string(
            const IT_Bus::QName& context_name
        ) = 0;

        virtual void
        register_context(
            const CORBAContextIdentifier& context_id,
            const IT_Bus::QName&          context_type
        ) = 0;

        virtual const IOP::ServiceId
        get_context_id(
            const IT_Bus::QName& context_name
        ) = 0;
```

Example 67: *The register_context() Function for CORBA Contexts*

```
    };
};
```

Writing and Reading Context Data

Overview

Figure 22 shows an overview of how context data instances are created in a multi-threaded application.

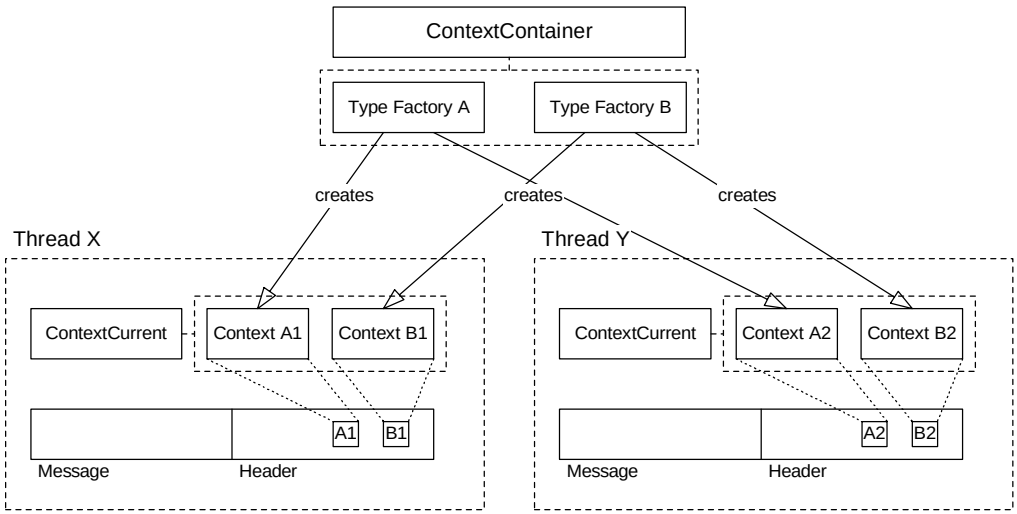

Figure 22: *Overview of Context Data in a Multi-Threaded Application*

Each application thread, for example X in Figure 22, is associated with its own context data instances, A1 and B1. Whenever an operation is invoked from a particular thread, the thread-specific context data is automatically inserted into the request message header.

Context current objects

A *context current* is an object that holds references to thread-specific context data. In particular, a context current holds reference to the context data instances used for sending and receiving context data.

Writing thread-specific context data to a request

Context data is initialized on a per-thread basis. Once you have initialized the context data for a particular thread, the context data is included in all request messages sent from this thread (but not in reply messages).

To write thread-specific context data, program the following steps:

1. Call the context container's `get_current()` function to obtain a reference to the context current object for this thread.

   You must also dynamically cast the returned `IT_Bus::ContextCurrent` object to the appropriate derived type (for example, `SoapContextCurrent` or `CORBAContextCurrent`).
2. Call the current object's `get_context()` function to obtain a concrete instance of a context data type. The returned context data instance is specific to the current thread.

   You must dynamically cast the returned `IT_Bus::AnyType` object to the context data type.
3. Initialize the context data instance. This context data will be included in all subsequent operation requests invoked from the current thread.

Reading context data from an incoming request

On the server side, you can access received context data within the scope of a called operation. Received context data is available only in a *calling context*; that is, in the context of servant code that services an incoming operation request. Without a calling context, the received context data is undefined.

To read context data from an incoming request, program the following steps:

1. Call the context container's `get_current()` function to obtain a reference to the context current object for this thread.

   You must also dynamically cast the returned `IT_Bus::ContextCurrent` object to the appropriate derived type (for example, `SoapContextCurrent` or `CORBAContextCurrent`).
2. Call the current object's `get_context()` function to obtain a reference to the received context data instance. The returned context data instance is specific to the current thread.

   You must dynamically cast the returned `IT_Bus::AnyType` object to the context data type.
3. Read the received context data using the type's member functions.

Context Example

Overview

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

In this section

This section contains the following subsections:

Custom SOAP Header Demonstration

Overview

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

ArtixInstallDir`/artix/`*Version*`/demos/advanced/custom_soap_header`

Figure 23 shows an overview of the custom SOAP header demonstration, showing how the client piggybacks context data along with an invocation request that is invoked on the `sayHi` operation.

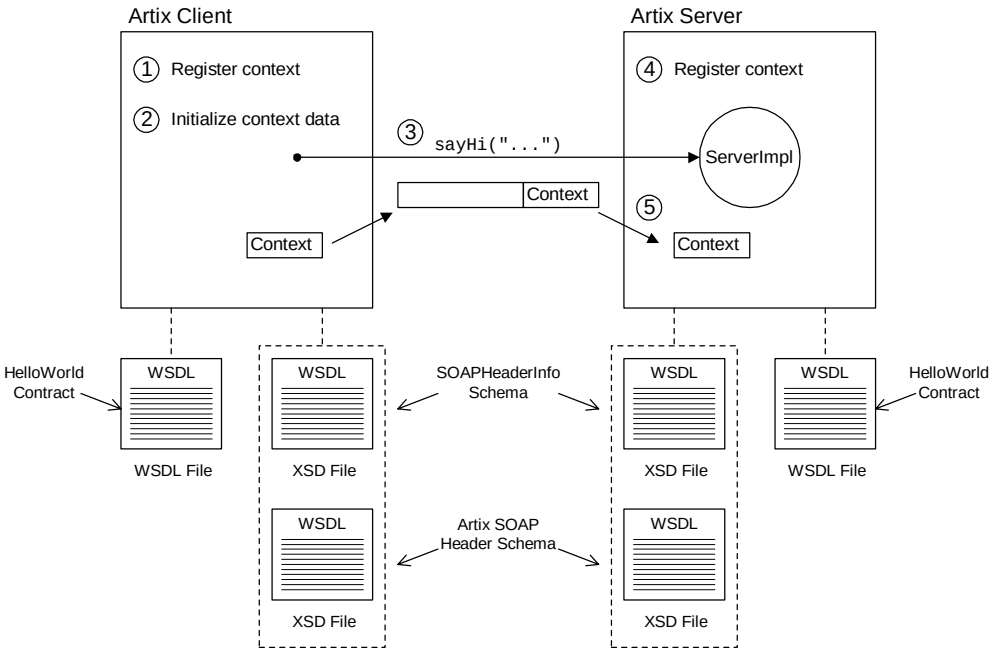

Figure 23: *Overview of the Custom SOAP Header Demonstration*

Transmission of context data

As illustrated in Figure 23, SOAP context data is transmitted as follows:

1. The client registers the context type, `SOAPHeaderInfo`, with the Bus.
2. The client initializes the context data instance.
3. The client invokes the `sayHi()` operation on the server.
4. As the server starts up, it registers the `SOAPHeaderInfo` context type with the Bus.
5. When the `sayHi()` operation request arrives on the server side, the `sayHi()` operation implementation extracts the context data from the request.

HelloWorld WSDL contract

The HelloWorld WSDL contract defines the contract implemented by the server in this demonstration. In particular, the HelloWorld contract defines the `Greeter` port type containing the `sayHi` WSDL operation.

SOAPHeaderInfo schema

The `SOAPHeaderInfo` schema (in the `demos/advanced/custom_soap_header/etc/contextTypes.xsd` file) defines the custom data type used as the context data type. This schema is specific to the custom SOAP header demonstration.

Artix SOAP header schema

The Artix SOAP header schema is used implicitly to define the overall header format for custom headers containing context data. This schema is generic to all Artix applications that send context data in a SOAP header.

Sample Context Schema

Overview

This subsection describes how to define an XML schema for a context type. In this example, the `SOAPHeaderInfo` type is declared in an XML schema. The `SOAPHeaderInfo` type is then used by the custom SOAP header demonstration to send custom data in a SOAP header.

SOAPHeaderInfo XML declaration

Example 68 shows the schema for the `SOAPHeaderInfo` type, which is defined specifically for the custom SOAP header demonstration to carry some sample data in a SOAP header. Note that Example 68 is a pure schema declaration, *not* a WSDL declaration.

Example 68: *XML Schema for the SOAPHeaderInfo Context Type*

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://schemas.iona.com/types/context"
    elementFormDefault="qualified"
   attributeFormDefault="unqualified">
    <xs:complexType name="SOAPHeaderInfo">
        <xs:annotation>
            <xs:documentation>
                Content to be added to a SOAP header
            </xs:documentation>
        </xs:annotation>
        <xs:sequence>
            <xs:element name="originator" type="xs:string"/>
            <xs:element name="message" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

The `SOAPHeaderInfo` complex type defines two member elements, as follows:

- `originator`—holds an arbitrary client identifier.
- `message`—holds an arbitrary example message.

Target namespace

You can use any target namespace for a context schema (as long as it does not clash with an existing namespace). This demonstration uses the following target namespace:

```
http://schemas.iona.com/types/context
```

Compiling the SOAPHeaderInfo schema

To compile the `SOAPHeaderInfo` schema, invoke the `wsdltocpp` compiler utility at the command line, as follows:

```
wsdltocpp contextTypes.xsd
```

Where `contextTypes.xsd` is a file containing the XML schema from Example 68. This command generates the following C++ stub files:

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

SOAPHeaderInfo C++ mapping

Example 69 shows how the schema from Example 68 on page 174 maps to C++, to give the `soap_interceptor::SOAPHeaderInfo` C++ class.

Example 69: *C++ Mapping of the SOAPHeaderInfo Context Type*

```
// C++
...
namespace soap_interceptor
{
    ...
    class SOAPHeaderInfo : public IT_Bus::SequenceComplexType
    {
      public:
        static const IT_Bus::QName type_name;

        SOAPHeaderInfo();
        SOAPHeaderInfo(const SOAPHeaderInfo & copy);
        virtual ~SOAPHeaderInfo();
        ...
        IT_Bus::String &       getoriginator();
        const IT_Bus::String & getoriginator() const;
        void setoriginator(const IT_Bus::String & val);

        IT_Bus::String &       getmessage();
        const IT_Bus::String & getmessage() const;
        void setmessage(const IT_Bus::String & val);
        ...
```

Example 69: *C++ Mapping of the SOAPHeaderInfo Context Type*

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

Client Main Function

Overview

This subsection discusses the client for the custom SOAP header demonstration. This client is designed to send a custom header, of `SOAPHeaderInfo` type, every time it invokes an operation on the `Greeter` port type.

To enable the sending of context data, the client performs two fundamental tasks, as follows:

1. *Register a context type with the SOAP container*—registering the context type is a prerequisite for sending context data in a request. By registering the context type with the Bus, you give the Bus instance the capability to marshal and unmarshal context data of that type.
2. *Initialize the context data in the SOAP current object*—before invoking any operations, the client obtains an instance of the context data from a SOAP current object. After initializing the context data, any operations invoked from the current thread will include the context data.

Client main function

Example 70 shows sample code from the client main function, which shows how to register a context type and initialize context data for the current thread.

Example 70: *Client Main Function Setting a SOAP Context*

```
// C++
// GreeterClientSample.cxx File

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

// Include header files related to the soap context
1 #include <it_bus_pdk/soap_context_container.h>
#include <it_bus/context_exception.h>

// Include header files representing the soap header content
2 #include "contextTypes_xsdTypes.h"
#include "contextTypes_xsdTypesFactory.h"
```

Example 70: *Client Main Function Setting a SOAP Context*

```
#include "GreeterClient.h"

IT_USING_NAMESPACE_STD

using namespace soap_interceptor;
using namespace IT_Bus;

int
main(int argc, char* argv[])
{
    try
    {
        IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);
        GreeterClient client;

3       ContextContainer* container =
            (bus->get_context_container("SoapContextContainer"));

        // Create QName objects needed to define a context
4       const QName principal_ctx_type(
            "",
            "SOAPHeaderInfo",
            "http://schemas.iona.com/types/context"
        );
5       const QName principal_message_name(
            "soap_header",
            "header_content",
            "http://schemas.iona.com/custom_header"
        );
        const String principal_part_name("header_info");

6       SoapContextContainer* soap_container =
            dynamic_cast<SoapContextContainer*> (container);

7       soap_container->register_context(
            principal_ctx_type,
            principal_message_name,
            principal_part_name
        );

8       SoapContextCurrent& soap_current =
   dynamic_cast<SoapContextCurrent&> (container->get_current());

9       AnyType& info = soap_current.get_context(
```

Example 70: *Client Main Function Setting a SOAP Context*

```
            principal_ctx_type,
            principal_message_name,
            principal_part_name
        );

10      SOAPHeaderInfo& header_info =
            dynamic_cast<SOAPHeaderInfo&> (info);

        const String originator("IONA Technologies");
        const String message("Artix is Powerful!");

        // Add the header content
        header_info.setoriginator(originator);
        header_info.setmessage(message);

        // Invoke the Web service business methods
        String theResponse;

11      client.sayHi(theResponse);
        cout << "sayHi response: " << theResponse << endl;
    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Error : Unexpected error occured!"
             << endl << e.message()
             << endl;
        return -1;
    }
    return 0;
}
```

The preceding code example can be explained as follows:

1. The `it_bus_pdk/soap_context_container.h` header file contains the declarations of the following classes:
   - `IT_Bus::SoapContextContainer` and,
   - `IT_Bus::SoapContextCurrent`.
2. The `contextTypes_xsdTypes.h` local header file contains the declaration of the `SOAPHeaderInfo` class, which has been generated from the context schema (see Example 68 on page 174).

3. The `IT_Bus::get_context_container()` function is called with the string argument, `SoapContextContainer`, to get an initial reference to an `IT_Bus::SoapContextContainer` object. You can, in principle, use `get_context_container()` to obtain references to context containers for a variety of different bindings.
4. The QName with namespace URI, `http://schemas.iona.com/types/context`, and local part, `SOAPHeaderInfo`, identifies the context type from Example 68 on page 174.
5. The QName with namespace URI, `http://schemas.iona.com/custom_header`, and local part, `header_content`, identifies the generic type of SOAP header that Artix uses to encapsulate context data.
6. The context container must be cast from its base type, `IT_Bus::ContextContainer`, to the derived type, `IT_Bus::SoapContextContainer`, in order to access the SOAP specific methods on this object.
7. This call to `register_context()` tells the Artix Bus that the `SOAPHeaderInfo` type will be used to send context data in SOAP headers. After you have registered the context, the Bus is prepared to marshal the context data (if any) into a SOAP header.
8. Call `IT_Bus::ContextContainer::get_current()` to obtain a reference to the `IT_Bus::SoapContextCurrent` object and cast it to the derived type. The current object is needed in order to initialize the context data that will accompany all operation requests originating from the current thread.
9. This `SoapContextCurrent::get_context()` call returns a thread-specific instance of a `SOAPHeaderInfo` object. Because multiple context types could be registered against a SOAP header, it is necessary to specify the exact type details in the arguments to `get_context()`.
10. The `IT_Bus::AnyType` class is the base type for all complex types in Artix. In this case, you can cast the `AnyType` instance, `info`, to its derived type, `SOAPHeaderInfo`.

By setting the `originator` and `message` elements of this `SOAPHeaderInfo` object, you are effectively fixing the context data for all operations invoked from this thread.

11. When you invoke the `sayHi()` operation, the context data is included in the SOAP header. From this point on, any WSDL operation invoked from the current thread will include the `SOAPHeaderInfo` context data in its SOAP header.

Server Main Function

Overview

This subsection discusses the main function for the server in the custom SOAP header demonstration. In addition to the usual boilerplate code for an Artix server (that is, registering a servant and calling `IT_Bus::run()`), this server also registers a context type with the Bus.

By registering a context type with the Bus, you give the Bus instance the capability to unmarshal context data of that type. This unmarshalling capability is then exploited in the implementation of the `sayHi()` operation (see Example 72 on page 185).

Server main function

Example 71 shows sample code from the server main function, which registers the `SOAPHeaderInfo` context type and then creates and registers a `GreeterImpl` servant object.

Example 71: *Server Main Function Registering a SOAP Context*

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

1   #include <it_bus_pdk/soap_context_container.h>

    #include "GreeterImpl.h"

    IT_USING_NAMESPACE_STD

    using namespace soap_interceptor;
    using namespace IT_Bus;

    int
    main(int argc, char* argv[])
    {
        try
        {
            IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);

2           ContextContainer* container =
                (bus->get_context_container("SoapContextContainer"));
```

Example 71: *Server Main Function Registering a SOAP Context*

```
3         SoapContextContainer* soap_container =
              dynamic_cast<SoapContextContainer*> (container);

4         const QName principal_ctx_type(
              "",
              "SOAPHeaderInfo",
              "http://schemas.iona.com/types/context"
          );
5         const QName principal_message_name(
              "soap_header",
              "header_content",
              "http://schemas.iona.com/custom_header"
          );
          const String principal_part_name("header_info");

6         soap_container->register_context(
              principal_ctx_type,
              principal_message_name,
              principal_part_name
          );

          GreeterImpl servant(bus);

          IT_Bus::QName service_name("", "SOAPService",
    "http://www.iona.com/custom_soap_interceptor");

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

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

The preceding code example can be explained as follows:

1. The `it_bus_pdk/soap_context_container.h` header file contains the declarations of the following classes:
   - `IT_Bus::SoapContextContainer` and,
   - `IT_Bus::SoapContextCurrent`.
2. The `IT_Bus::get_context_container()` function is called with the string argument, `SoapContextContainer`, to get an initial reference to an `IT_Bus::SoapContextContainer` object. You can, in principle, use `get_context_container()` to obtain references to context containers for a variety of different bindings.
3. The context container must be cast from its base type, `IT_Bus::ContextContainer`, to the derived type, `IT_Bus::SoapContextContainer`, in order to access the SOAP specific methods on this object.
4. The QName with namespace URI, `http://schemas.iona.com/types/context`, and local part, `SOAPHeaderInfo`, identifies the context type from Example 68 on page 174.
5. The QName with namespace URI, `http://schemas.iona.com/custom_header`, and local part, `header_content`, identifies the generic type of SOAP header that Artix uses to encapsulate context data.
6. This call to `register_context()` tells the Artix Bus that the `SOAPHeaderInfo` type will be used to receive context data in SOAP headers. After you have registered the context, the Bus is prepared to unmarshal the context data (if any) from a SOAP header.

Service Implementation

Overview

This subsection discusses the implementation of the `Greeter` port type, which maps to the `GreeterImpl` servant class in C++.

In the custom SOAP header demonstration, the `GreeterImpl::sayHi()` operation is modified to peek at the context data accompanying the invocation. To access the context data, you need to get access to a context current object, which encapsulates all of the context data received from the client.

Implementation of the sayHi operation

Example 72 shows the implementation of the `sayHi()` operation from the `GreeterImpl` servant class. The `sayHi()` operation implementation uses the context API to access the context data received from the client.

Example 72: *sayHi Operation Accessing a SOAP Context*

```
// C++
...
void
GreeterImpl::sayHi(
    IT_Bus::String &theResponse
) IT_THROW_DECL((IT_Bus::Exception))
{
    cout << "sayHi invoked" << endl;
    theResponse = "Hello from Artix";

    Bus_var bus = Bus::create_reference();

1   ContextContainer* container =
        (bus->get_context_container("SoapContextContainer"));

2   SoapContextCurrent& soap_current =
   dynamic_cast<SoapContextCurrent&> (container->get_current());

3   const QName principal_ctx_type(
        "",
        "SOAPHeaderInfo",
        "http://schemas.iona.com/types/context"
    );
3   const QName principal_message_name(
        "soap_header",
```

Example 72: *sayHi Operation Accessing a SOAP Context*

```
        "header_content",
        "http://schemas.iona.com/custom_header"
    );
    const String principal_part_name("header_info");

4   AnyType& info = soap_current.get_context(
        principal_ctx_type,
        principal_message_name,
        principal_part_name
    );

5   SOAPHeaderInfo& header_info =
        dynamic_cast<SOAPHeaderInfo&> (info);

6   // Extract the application specific SOAP header information
    String& originator = header_info.getoriginator();
    String& message = header_info.getmessage();

    cout << "SOAP Header originator = "
         << originator.c_str() << endl;
    cout << "SOAP Header message = " << message.c_str() << endl;
}
```

The preceding code example can be explained as follows:

1. The `IT_Bus::get_context_container()` function is called with the string argument, `SoapContextContainer`, to get an initial reference to an `IT_Bus::SoapContextContainer` object.
2. Call `IT_Bus::ContextContainer::get_current()` to obtain a reference to the `IT_Bus::SoapContextCurrent` object and cast it to the derived type. The current object provides access to the context data received from the client (if any).
3. The QName with namespace URI, `http://schemas.iona.com/types/context`, and local part, `SOAPHeaderInfo`, identifies the context type from Example 68 on page 174.
4. The QName with namespace URI, `http://schemas.iona.com/custom_header`, and local part, `header_content`, identifies the generic type of SOAP header that Artix uses to encapsulate context data.

5. The `IT_Bus::AnyType` class is the base type for all complex types in Artix. In this case, you can cast the `AnyType` instance, `info`, to its derived type, `SOAPHeaderInfo`.
6. You can now access the context data by calling the accessors for the `originator` and `message` elements, `getoriginator()` and `getmessage()`.

CHAPTER 8

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 24 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.

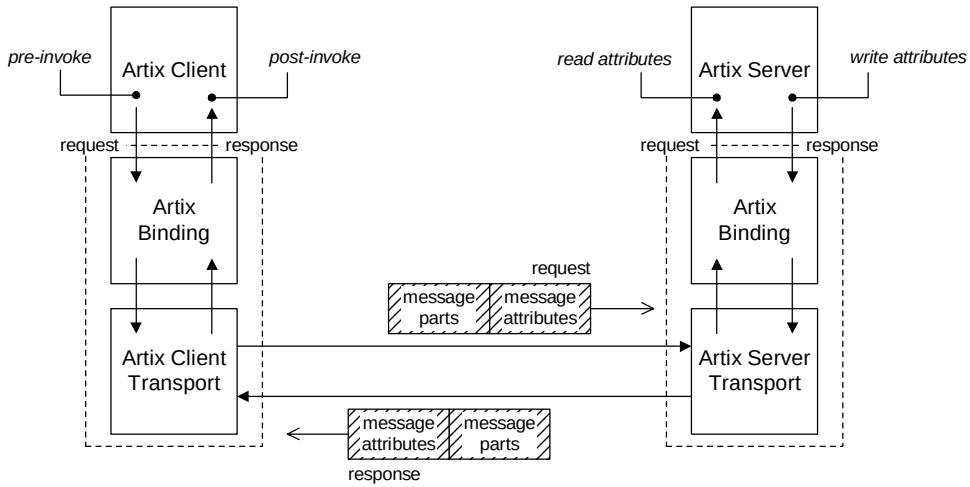

Figure 24: *Passing Message Attributes in Input and Output Messages*

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/transports/soap_over_mq) 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 195.
- "Transport-Specific API" on page 199.

Schemas

Overview

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

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

Schema documentation

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

Schemas for message attributes

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

Table 4: *Transport Schemas with Message Attributes*

Schema Type	File
HTTP	*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 73 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 73: *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 193).

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

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

Inheritance hierarchy

Figure 25 shows the inheritance hierarchy for the classes involved in the name-value API for message attributes.

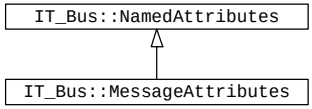

Figure 25: *Inheritance Hierarchy for IT_Bus::MessageAttributes Class*

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 74, 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 74: *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 74: *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 74: *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 26 shows the inheritance hierarchy for the classes involved in the transport-specific API for message attributes.

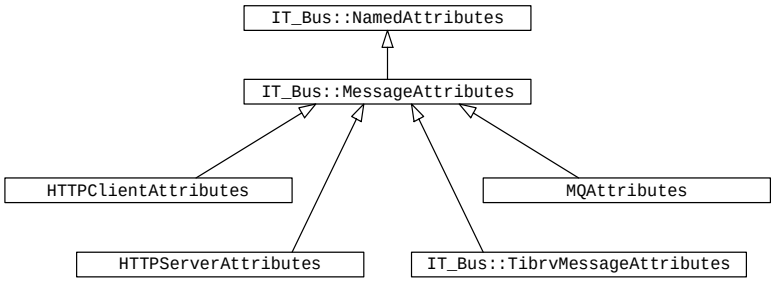

Figure 26: *Inheritance Hierarchy for the Transport-Specific API*

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 75, 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 75: *Getting and Setting Tibco Message Attributes*

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

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

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

        virtual const ULong& get_sequence();

        virtual const Double& get_time_limit();
```

Example 75: *Getting and Setting Tibco Message Attributes*

```
        virtual void set_time_limit(const Double& time_limit);

        virtual const UByte& get_jms_delivery_mode();

        virtual const UByte& get_jms_priority();

        virtual const ULong& get_jms_timestamp();

        virtual const ULong& get_jms_expiration();

        virtual const String& get_jms_type();

        virtual const String& get_jms_message_id();

        virtual const String& get_jms_correlation_id();

        virtual const Boolean& get_jms_redelivered();
        ...
    };
};
```

Using Message Attributes in a Client

Overview

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

How to use message attributes in a client

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

Step	Action
1	Obtain an `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 76. Edit the `client.cxx` file, which is located in the *ArtixInstallDir*`/artix/`*Version*`/demos/basic/hello_world_soap_http/cxx/client` directory. In Example 76, 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 76: *Using Message Attributes in a Client*

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

Example 76: *Using Message Attributes in a Client*

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

    HelloWorldClient hw;

    String string_in;
    String string_out;

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

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

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

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

The preceding client code example can be explained as follows:

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

   You switch on message attributes on the client side by calling `use_input_message_attributes()` and `use_output_message_attributes()` on the port object. By default, the message attribute feature is not enabled because it adds a certain performance penalty.
2. Pre-invoke interception point—the input message attribute object, `hw_input`, enables you to set attributes that are passed over the connection to the server.
3. The `sayHi()` operation performs the remote procedure call on the server.
4. Post-invoke interception point—the output message attribute object, `hw_output`, enables you to retrieve the attributes sent by the server.
5. The `IT_Bus::NoSuchAttributeException` exception is thrown if you try to read an output attribute that was not sent by the server.

Using Message Attributes in a Server

Overview

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

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

How to use message attributes in a server

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

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

C++ example

To use message attributes in a server, you can modify the HelloWorld HTTP SOAP server as shown in Example 77. Edit the `HelloWorldImpl.cxx` file, which is located in the *ArtixInstallDir*`/artix/`*Version*`/demos/basic/hello_world_soap_http/cxx/server` directory. In Example 77, 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 77: *Using Message Attributes in a Server*

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

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

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

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

Example 77: *Using Message Attributes in a Server*

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

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

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

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

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

The server code in Example 77 can be explained as follows:

1. In the `HelloWorldImpl` constructor, call `use_input_message_attributes()` and `use_output_message_attributes()` on the port object to initialize the message attribute functionality.
2. The servant's `Current` object is obtained through the `Bus` object representing the server connection. The `get_bus()` operation is defined on the `IT_Bus::ServerStubBase` class, which is a base class of `HelloWorldImpl`. It returns a reference to the `Bus` object that represents the server connection.

3. The `get_port()` operation is defined on the `IT_Bus::Operation` class, which is accessed through the current object's `get_operation()` operation.

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

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

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

CHAPTER 9

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:QName`	`IT_Bus::QName` *(SOAP only)*
`xsd:dateTime`	`IT_Bus::DateTime`
`xsd:decimal`	`IT_Bus::Decimal`
`xsd:base64Binary`	`IT_Bus::BinaryBuffer`
`xsd:hexBinary`	`IT_Bus::BinaryBuffer`

String Type

Overview

The `xsd:string` type maps to `IT_Bus::String`, which is typedef'ed in `it_bus/ustring.h` to `IT_Bus::IT_UString` class. For a full definition of `IT_Bus::String`, see `it_bus/ustring.h`.

IT_Bus::String class

The `IT_Bus::String` class is modelled on the standard ANSI string class. Hence, the `IT_Bus::String` class overloads the + and += operators for concatenation, the `[]` operator for indexing characters, and the ==, `!=`, >, <, >=, <= operators for comparisons.

String iterator class

The corresponding string iterator class is `IT_Bus::String::iterator`.

C++ example

The following C++ example shows how to perform some basic string manipulation with `IT_Bus::String`:

```
// C++
IT_Bus::String s = "A C++ ANSI string."
s += " And here is some string concatenation."

// Now convert to a C style string.
// (Note: s retains ownership of the memory)
const char *p = s.c_str();
```

Internationalization

The `IT_Bus::String` class supports the use of international characters. When using international characters, you should configure your Artix application to use a particular code set by editing the Artix domain configuration file, `artix.cfg`. The configuration details depend on the type of Artix binding, as follows:

- SOAP binding—set the `plugins:soap:encoding` configuration variable.
- CORBA binding—set the `plugins:codeset:char:ncs`, `plugins:codeset:char:ccs`, `plugins:codeset:wchar:ncs`, and `plugins:codeset:wchar:ccs` configuration variables.

For more details about configuring internationalization, see the "Using Artix with International Codesets" chapter of the *Deploying and Managing Artix Solutions* document.

Encoding arguments

Some of the `IT_Bus::String` functions take an optional string argument, `encoding`, that lets you specify a character set encoding for the string.

The `encoding` argument must be a standard IANA character set name. For example, Table 6 shows some of commonly used IANA character set names:

Table 6: *IANA Character Set Names*

IANA Name	Description
US-ASCII	7-bit ASCII for US English.
ISO-8859-1	Western European languages.
UTF-8	Byte oriented transformation of Unicode.
UTF-16	Double-byte oriented transformation of 4-byte Unicode.
Shift_JIS	Japanese DOS & Windows.
EUC-JP	Japanese adaptation of generic EUC scheme, used in UNIX.
EUC-CN	Chinese adaptation of generic EUC scheme, used in UNIX.
ISO-2022-JP	Japanese adaptation of generic ISO 2022 encoding scheme.
ISO-2022-CN	Chinese adaptation of generic ISO 2022 encoding scheme.
BIG5	Big Five is a character set developed by a consortium of five companies in Taiwan in 1984.

Artix supports all of the character sets defined in International Components for Unicoded (ICU) 2.6. For a full listing of supported character sets, see http://www-124.ibm.com/icu/index.html (part of the IBM open source project http://oss.software.ibm.com).

Constructors

The `IT_Bus::String` class defines a default constructor and non-default constructors to initialize a string using narrow and wide characters, as follows:

- Narrow character constructors.
- 16-bit character constructor.
- wchar_t character constructor.

Narrow character constructors

Example 78 shows three different constructors that can be used to initialize an `IT_UString` with a narrow character string.

Example 78: *Narrow Character Constructors*

```
IT_UString(
    const char*          str,
    size_t               n  = npos,
    const char*          encoding = 0,
    IT_ExceptionHandler& eh = IT_EXCEPTION_HANDLER
);
IT_UString(
    size_t               n,
    char                 c,
    const char*          encoding = 0,
    IT_ExceptionHandler& eh = IT_EXCEPTION_HANDLER
);
IT_UString(
    const IT_String&     s,
    size_t               pos = 0,
    size_t               n   = npos,
    const char*          encoding = 0,
    IT_ExceptionHandler& eh  = IT_EXCEPTION_HANDLER
);
```

The constructor signatures are similar to the standard ANSI string constructors, except for the additional `encoding` argument. A null `encoding` argument, `encoding=0`, implies the constructor uses the local character set.

16-bit character constructor

Example 79 shows the constructor that can be used to initialize an `IT_UString` with an array of 16-bit characters (represented by `unsigned short*`).

Example 79: *16-Bit Character Constructor*

```
IT_UString(
    const unsigned short* sb,
    const IT_String&      encoding,
    size_t                n = npos,
    IT_ExceptionHandler&  eh = IT_EXCEPTION_HANDLER
);
```

wchar_t character constructor

Example 80 shows the constructor that can be used to initialize an `IT_UString` with an array of `wchar_t` characters.

Example 80: *wchar_t Character Constructor*

```
IT_UString(
    const wchar_t*        wb,
    size_t                n = npos,
    IT_ExceptionHandler&  eh = IT_EXCEPTION_HANDLER
);
```

String conversion functions

The member functions shown in Example 81 are used to convert an `IT_Bus::String` to an ordinary C-style string, a UTF-16 format string and a `wchar_t` format string:

Example 81: *String Conversion Functions*

```
// C++
const char* c_str(
    const char* encoding = 0
) const;  // has NUL character at end

const unsigned short* utf16_str() const;

const wchar_t*        wchar_t_str() const;
```

If you want to copy the return value from a string conversion function, you also need to know the dimension of the relevant array. For this, you can use the `IT_Bus::String::length()` function:

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

The `IT_Bus::String::length()` function returns the number of underlying characters in a string, irrespective of how many bytes it takes to represent each character. Hence, the size of the array required to hold a copy of a converted string equals `length()+1` (an extra array element is required for the `NUL` character).

String conversion examples

Example 82 shows you how to convert and copy a string, `s`, into a C-style string, a UTF-16 format string and a `wchar_t` format string.

Example 82: *String Conversion Examples*

```
// C++
// Copy 's' into a plain 'char *' string:
char *s_copy = new char[s.length()+1];
strcpy(s_copy, s.c_str());

// Copy 's' into a UTF-16 string:
unsigned short* utf16_copy = new unsigned short[s.length()+1];
const unsigned short* utf16_p = s.utf16_str();
for (i=0; i<s.length()+1; i++) {
    utf16_copy[i] = utf16_p[i];
}

// Copy 's' into a wchar_t string:
wchar_t* wchar_t_copy = new wchar_t[s.length()+1];
const wchar_t* wchar_t_p = s.wchar_t_str();
for (i=0; i<s.length()+1; i++) {
    wchar_t_copy[i] = wchar_t_p[i];
}
```

Reference

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

For more details about internationalization in Artix, see the "Using Artix with International Codesets" chapter of the *Deploying and Managing Artix Solutions* document.

QName Type

Overview

`xsd:QName` maps to `IT_Bus::QName`. A qualified name, or QName, is the unique name of a tag appearing in an XML document, consisting of a *namespace URI* and a *local part*.

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

QName constructor

The usual way to construct an `IT_Bus::QName` object is by calling the following constructor:

```
// C++
QName::QName(
  const String &    namespace_prefix,
  const String &    local_part,
  const String &    namespace_uri
)
```

Because the namespace prefix is relatively unimportant, you can leave it blank. For example, to create a QName for the `<soap:address>` element:

```
// C++
IT_Bus::QName soap_address = new IT_Bus::QName(
    "",
    "address",
    "http://schemas.xmlsoap.org/wsdl/soap"
);
```

QName member functions

The `IT_Bus::QName` class has the following public member functions:

```
const IT_Bus::String &
get_namespace_prefix() const;

const IT_Bus::String &
get_local_part() const;

const IT_Bus::String &
get_namespace_uri() const;

const IT_Bus::String get_raw_name() const;
const IT_Bus::String to_string() const;
```

```
bool has_unresolved_prefix() const;
size_t get_hash_code() const;
```

QName equality

The == operator can be used to test for equality of `IT_Bus::QName` objects. QNames are tested for equality as follows:

1. Assuming that a namespace URI is defined for the QNames, the QNames are equal if their namespace URIs match and the local part of their element names match.
2. If one of the QNames lacks a namespace URI (empty string), the QNames are equal if their namespace prefixes match and the local part of their element names match.

Date and Time Types

Overview

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

Table 7: *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 8 for a list of supported operators.

Table 8: *Operators Supported by IT_Bus::Decimal*

Description	Operators
Arithmetical operators	`+, -, *, /, ++, --`
Assignment operators	`=, +=, -=, *=, /=`
Comparison operators	`==, !=, >, <, >=, <=`

IT_Bus::Decimal member functions

The following member functions are supported by `IT_Bus::Decimal`:

```
// C++
IT_Bus::Decimal round(unsigned short scale) const;

IT_Bus::Decimal truncate(unsigned short scale) const;

unsigned short number_of_digits() const;

unsigned short scale() const;

IT_Bool is_negative() const;

int compare(const IT_FixedPoint& val) const;

IT_Bus::Decimal::DigitIterator left_most_digit() const;
IT_Bus::Decimal::DigitIterator past_right_most_digit() const;
```

IT_Bus::Decimal::DigitIterator

The `IT_Bus::Decimal::DigitIterator` type is an ANSI-style iterator class that iterates over all the digits in a fixed point decimal instance.

C++ example

The following C++ example shows how to perform some elementary arithmetic using the `IT_Bus::Decimal` type.

```
// C++
IT_Bus::Decimal d1 = "123.456";
IT_Bus::Decimal d2 = "87654.321";

IT_Bus::Decimal d3 = d1+d2;
d3 *= d1;
if (d3 > 100000) {
   cout << "d3 = " << d3;
}
```

Binary Types

Overview

There are two WSDL binary types, which map to C++ as shown in Table 9:

Table 9: *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 83 shows how a server might implement the `echoHexBinary` operation.

Example 83: *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 84 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 84: *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 85. The only values that can legally be set using the `set_value()` member function are the strings `RED`, `GREEN`, or `BLUE`.

Example 85: *C++ Mapping of ColorEnum Restricted Type*

```
// C++
class ColorEnum : public IT_Bus::AnySimpleType
{
    ...
  public:
    ColorEnum();
    ColorEnum(const IT_Bus::String & value);
    ...

    ColorEnum& operator= (const ColorEnum& assign);
    IT_Bus::Boolean operator== (const ColorEnum& copy);

    virtual const IT_Bus::QName & get_type() const;
    void           set_value(const IT_Bus::String & value);
    IT_Bus::String get_value() const;
};
```

Unsupported Simple Types

List of unsupported simple types

The following WSDL simple types are currently not supported by the WSDL-to-C++ compiler:

Atomic Simple Types

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

Other Simple Types

`xsd:list`
`xsd:union`

Complex Types

Overview

This section describes the WSDL-to-C++ mapping for complex types. Complex types are defined within an XML schema. In contrast to simple types, complex types can contain elements and carry attributes.

In this section

This section contains the following subsections:

Sequence Complex Types

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 259.

WSDL example

Example 86 shows an example of a sequence, `SequenceType`, with three elements.

Example 86: *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 86) to the SequenceType C++ class. An outline of this class is shown in Example 87.

Example 87: *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 88 shows how a client could use a proxy instance, bc, to invoke the echoSequence operation.

Example 88: *Client Invoking an echoSequence Operation*

```
// C++
SequenceType seqIn, seqResult;
seqIn.setvarFloat(3.14159);
seqIn.setvarInt(54321);
seqIn.setvarString("You can use a string constant here.");

try {
    bc.echoSequence(seqIn, seqResult);

    if((seqResult.getvarInt() != seqIn.getvarInt()) ||
       (seqResult.getvarFloat() != seqIn.getvarFloat()) ||
      (seqResult.getvarString().compare(seqIn.getvarString()) !=
   0))
    {
        cout << endl << "echoSequence FAILED" << endl;
        return;
    }
} catch (IT_Bus::FaultException &ex)
{
    cout << "Caught Unexpected FaultException" << endl;
    cout << ex.get_description().c_str() << endl;
}
```

Choice Complex Types

Overview

XML schema choice complex types are mapped to a generated C++ class, which inherits from `IT_Bus::ChoiceComplexType`. The mapped C++ class is defined in the generated *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 89 shows an example of a choice complex type, `ChoiceType`, with three elements.

Example 89: *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 89) to the SequenceType C++ class. An outline of this class is shown in Example 90.

Example 90: *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 90: *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 91 shows how a client could use a proxy instance, `bc`, to invoke the `echoChoice` operation.

Example 91: *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 91: *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 92 shows an example of an all complex type, `AllType`, with three elements.

Example 92: *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 92) to the `AllType` C++ class. An outline of this class is shown in Example 93.

Example 93: *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 94 shows how a client could use a proxy instance, `bc`, to invoke the `echoAll` operation.

Example 94: *Client Invoking an echoAll Operation*

```
// C++
AllType allIn, allResult;
allIn.setvarFloat(3.14159);
allIn.setvarInt(54321);
allIn.setvarString("You can use a string constant here.");

try {
    bc.echoAll(allIn, allResult);

    if((allResult.getvarInt() != allIn.getvarInt()) ||
       (allResult.getvarFloat() != allIn.getvarFloat()) ||
      (allResult.getvarString().compare(allIn.getvarString()) !=
   0))
    {
        cout << endl << "echoAll FAILED" << endl;
        return;
    }
} catch (IT_Bus::FaultException &ex)
{
    cout << "Caught Unexpected FaultException" << endl;
    cout << ex.get_description().c_str() << endl;
}
```

Attributes

Overview

Artix supports the use of `<attribute>` declarations within the scope of a `<complexType>` definition. For example, you can include attributes in the definitions of an all complex type, sequence complex type, and choice complex type. The declaration of an attribute in a complex type has the following syntax:

```
<attribute name="AttrName" type="AttrType"
   use="[optional|required|prohibited]"/>
```

Attribute use

When declaring an attribute, the use can have one of the following values:

- `optional`—(default) the attribute can either be set or unset.
- `required`—the attribute must be set.
- `prohibited`—the attribute must be unset (cannot be used).

On-the-wire optimization

Artix optimizes the transmission of attributes by distinguishing between set and unset attributes. Only *set* attributes are transmitted (on bindings that support this optimization).

Note: The CORBA binding does not support this optimization.

C++ mapping overview

There are two different styles of C++ mapping for attributes, depending on the use value in the attribute declaration:

- *Optional attributes*—if an attribute is declared with `use="optional"` (or if the use setting is omitted altogether), the generated `get`*Attribute*`()` function returns a pointer, instead of a reference, to the attribute value. This enables you to test whether the the attribute is set or not by testing the pointer for nilness (whether it equals 0).
- *Required attributes*—if an attribute is declared with `use="required"`, the generated `get`*Attribute*`()` function returns a reference to the attribute value.

Optional attribute example

Example 95 shows how to define a sequence type with a single optional attribute, `prop`, of `xsd:string` type (attributes are optional by default).

Example 95: *Definition of a Sequence Type with an Optional Attribute*

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

C++ mapping for an optional attribute

Example 96 shows an outline of the C++ `SequenceType` class generated from Example 95, which defines accessor and modifier functions for the optional `prop` attribute.

Example 96: *Mapping an Optional Attribute to C++*

```
// C++
class SequenceType : public IT_Bus::SequenceComplexType
{
  public:
    SequenceType();
    ...
1   const IT_Bus::String * getprop() const;
    IT_Bus::String * getprop();

2   void setprop(const IT_Bus::String * val);
3   void setprop(const IT_Bus::String & val);
};
```

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

1. If the attribute is set, returns a pointer to its value; if not, returns `0`.
2. If `val != 0`, sets the attribute to `*val` (makes a copy); if `val == 0`, unsets the attribute.
3. Sets the attribute to `val` (makes a copy). This is a convenience function that enables you to set the attribute without using a pointer.

Required attribute example

Example 97 shows how to define a sequence type with a single required attribute, `prop`, of `xsd:string` type.

Example 97: *Definition of a Sequence Type with a Required Attribute*

```
<complexType name="SequenceType">
    <sequence>
        <element name="varFloat" type="xsd:float"/>
        <element name="varInt" type="xsd:int"/>
        <element name="varString" type="xsd:string"/>
    </sequence>
    <attribute name="prop" type="xsd:string" use="required"/>
</complexType>
```

C++ mapping for a required attribute

Example 98 shows an outline of the C++ `SequenceType` class generated from Example 97 on page 241, which defines accessor and modifier functions for the required `prop` attribute.

Example 98: *Mapping a Required Attribute to C++*

```
// C++
class SequenceType : public IT_Bus::SequenceComplexType
{
  public:
    SequenceType();
    ...
    const IT_Bus::String & getprop() const;
    IT_Bus::String & getprop();

    void setprop(const IT_Bus::String & val);
};
```

In this case, the `getprop()` accessor function returns a *reference* to a string (that is, `IT_Bus::String&`), rather than a pointer to a string.

Limitations

The following attribute types are *not* supported:

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

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 99.

WSDL example

Example 99 shows an example of a nested complex type, which features a choice complex type, `NestedChoiceType`, nested inside a sequence complex type, `SeqOfChoiceType`.

Example 99: *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 100 shows an outline of the generated C++ `NestedChoiceType` class.

Example 100: *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 101 shows an outline of the generated C++ `SeqOfChoiceType` class, which shows how the nested complex type is mapped within a sequence complex type.

Example 101: *Mapping of SeqOfChoiceType to C++*

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

Example 101:*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 94 shows how a client could use a proxy instance, `bc`, to invoke the `echoSeqOfChoice` operation.

Example 102:*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 102:*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 103 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 103: *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 104 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 104: *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 105 shows an outline of the C++ `internationalPrice` class generated from Example 104 on page 248.

Example 105: *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 105:*Mapping the internationalPrice Type to C++*

```
    const IT_Bus::Decimal & get_simpleTypeValue() const;
    IT_Bus::Decimal & get_simpleTypeValue();
    void set_simpleTypeValue(const IT_Bus::Decimal & val);
    ...
};
```

The value of the currency attribute, which is added by extension, can be accessed and modified using the `getcurrency()` and `setcurrency()` member functions. The simple type value (that is, the value enclosed between the `<internationalPrice>` and `</internationalPrice>` tags) can be accessed and modified by the `get_simpleTypeValue()` and `set_simpleTypeValue()` member functions.

Deriving a Complex Type from a Complex Type

Overview

Artix supports derivation of a complex type from a complex type, for which the following kinds of derivation are possible:

- Derivation by restriction—currently *not* supported by Artix.
- Derivation by extension.

This subsection describes the C++ mapping for complex types derived from complex types and, in particular, describes the coding pattern for calling a function either with base type arguments or with derived type arguments.

Allowed inheritance relationships

Figure 27 shows the inheritance relationships allowed between complex types. As well as inheriting between the same kind of complex type (sequence from sequence, choice from choice, and all from all), it is possible to cross-inherit. For example, a sequence can derive from a choice, a choice from an all, an all from a choice, and so on.

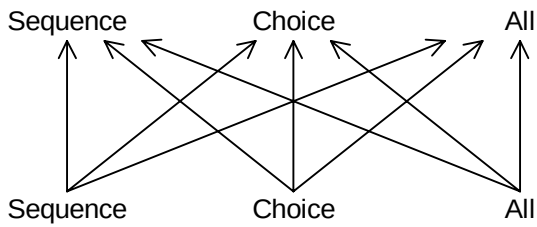

Figure 27: *Allowed Inheritance Relationships for Complex Types*

Derivation by extension

Example 106 shows an example of deriving a sequence from a sequence by extension. In this example, `DerivedStruct_BaseStruct` is derived from `SimpleStruct` by extension. The standard tag used to declare inheritance by extension is `<extension base="`*BaseComplexType*`"/>`.

Example 106: *Example of Deriving a Sequence by Extension*

```
<complexType name="SimpleStruct">
    <sequence>
        <element name="varFloat" type="float"/>
        <element name="varInt" type="int"/>
        <element name="varString" type="string"/>
    </sequence>
    <attribute name="varAttrString" type="string"/>
</complexType>
...
1   <complexType name="DerivedStruct_BaseStruct">
2       <complexContent mixed="false">
3           <extension base="tns:SimpleStruct">
4               <sequence>
                    <element name="varStringExt" type="string"/>
                    <element name="varFloatExt" type="float"/>
                </sequence>
5               <attribute name="attrString1" type="string"/>
            </extension>
        </complexContent>
6       <attribute name="attrString2" type="string"/>
</complexType>
```

The preceding type definition can be explained as follows:

1. This `<complexType>` tag introduces the definition of the derived sequence type, `DerivedStruct_BaseStruct`.
2. The `<complexContent>` tag indicates that what follows is a declaration of contained tags. The `mixed="false"` setting indicates that the type can contain only tags, not text.
3. The `<extension>` tag indicates that this type derives by extension from the `SimpleStruct` type.
4. The `<sequence>` tag defines extra type members that are specific to the derived type, `DerivedStruct_BaseStruct`.
5. You can also declare attributes specific to the derived type.

6. Attributes can also be declared directly within the scope of `<complexType>`.

C++ mapping

The sequence types defined in Example 106 on page 251, `SimpleStruct` and `DerivedStruct_BaseStruct`, map to C++ as shown in Example 107.

Example 107: *C++ Mapping of a Derived Sequence Type*

```
// C++
class SimpleStruct : public IT_Bus::SequenceComplexType
{
  public:
    static const IT_Bus::QName type_name;

    SimpleStruct();
    ...
    IT_Bus::AnyType &
    operator=(const IT_Bus::AnyType & rhs);

    SimpleStruct &
    operator=(const SimpleStruct & rhs);

    const SimpleStruct * get_derived() const;
    virtual IT_Bus::AnyType::Kind get_kind() const;
    virtual const IT_Bus::QName & get_type() const;
    ...
    IT_Bus::Float       getvarFloat();
    const IT_Bus::Float getvarFloat() const;
    void setvarFloat(const IT_Bus::Float val);

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

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

    IT_Bus::String &       getvarAttrString();
    const IT_Bus::String & getvarAttrString() const;
    void setvarAttrString(const IT_Bus::String & val);

  private:
    ...
};
```

Example 107: *C++ Mapping of a Derived Sequence Type*

```
typedef IT_AutoPtr<SimpleStruct> SimpleStructPtr;

...
class IT_TEST_WSDL_API DerivedStruct_BaseStruct : public
   SimpleStruct , public virtual
   IT_Bus::ComplexContentComplexType
{
  public:
    static const IT_Bus::QName type_name;

    DerivedStruct_BaseStruct();
    DerivedStruct_BaseStruct(const DerivedStruct_BaseStruct &
   copy);
    virtual ~DerivedStruct_BaseStruct();
    ...
    IT_Bus::String &       getvarStringExt();
    const IT_Bus::String & getvarStringExt() const;
    void setvarStringExt(const IT_Bus::String & val);

    IT_Bus::Float          getvarFloatExt();
    const IT_Bus::Float    getvarFloatExt() const;
    void setvarFloatExt(const IT_Bus::Float val);

    IT_Bus::String &       getattrString1();
    const IT_Bus::String & getattrString1() const;
    void setattrString1(const IT_Bus::String & val);

    IT_Bus::String &       getattrString2();
    const IT_Bus::String & getattrString2() const;
    void setattrString2(const IT_Bus::String & val);

  private:
    ...
};
```

The C++ `DerivedStruct_BaseStruct` class derives directly from the C++ `SimpleStruct` class. Hence, all of the accessors and modifiers declared in the base class, `SimpleStruct`, are also available to the derived class, `DerivedStruct_BaseStruct`.

Using a base type as a holder

The `SimpleStruct` type declared in Example 107 on page 252 is really a dual-purpose type. That is, a `SimpleStruct` instance can be used in one of the following different ways:

- As a `SimpleStruct` data type (base type)—member data is accessed by invoking `get`*ElementName*`()` and `set`*ElementName*`()` functions directly on the `SimpleStruct` instance.
- As a holder type (derived type holder)—in this usage pattern, the `SimpleStruct` instance is used to hold a reference to a more derived type (for example, `DerivedStruct_BaseStruct`).

Holder type functions

If you are using `SimpleStruct` as a holder type, the following member functions are relevant:

- `SimpleStruct(const SimpleStruct & copy)`—the `SimpleStruct` copy constructor is used to initialize the reference held by the `SimpleStruct` holder object. The type passed to the copy constructor can be any type derived from `SimpleStruct`.
- `SimpleStruct & operator=(const SimpleStruct & rhs)`—alternatively, if you already have a `SimpleStruct` object, you can change the reference held by making an assignment to the `SimpleStruct` holder.
- `const SimpleStruct * get_derived() const`—if you want to access the derived type held by a `SimpleStruct` holder object, call the `get_derived()` member function and then dynamically cast the return value to the appropriate type.
- `const IT_Bus::QName & get_type() const`—call `get_type()` to get the QName of the derived type held by a `SimpleStruct` holder object.

Polymorphism

When a WSDL operation is defined to take arguments of a base class type (for example, `SimpleStruct`), it is also possible to send and receive arguments of a type derived from that base class (for example, `DerivedStruct_BaseStruct`).

For reasons of backward compatibility, however, the C++ code required for calling an operation with derived type arguments is different from the C++ code required for calling an operation with base type arguments.

Sample WSDL operation

For example, consider the definition of the following WSDL operation, `test_SimpleStruct`, that takes an *in* argument of `SimpleStruct` type and returns an *out* argument of `SimpleStruct` type.

Example 108: *The test_SimpleStruct Operation with Base Type Arguments*

```
...
<message name="test_SimpleStruct">
    <part name="x" element="tns:SimpleStruct_x"/>
</message>
<message name="test_SimpleStruct_response">
    <part name="return" element="tns:SimpleStruct_return"/>
</message>
...
<operation name="test_SimpleStruct">
  <input name="test_SimpleStruct"
         message="tns:test_SimpleStruct"/>
  <output name="test_SimpleStruct_response"
          message="tns:test_SimpleStruct_response"/>
</operation>
```

The preceding `test_SimpleStruct` WSDL operation maps to the following C++ function (in the `TypeTestClient` client proxy class).

```
// C++
virtual void
test_SimpleStruct(
    const SimpleStruct &x,
    SimpleStruct &_return,
) IT_THROW_DECL((IT_Bus::Exception));
```

To call the preceding `test_SimpleStruct()` function in C++, use one of the following programming patterns, depending on the type of arguments passed:

- Base or derived type arguments.
- Base type arguments only (for legacy code).

Base or derived type arguments

Example 109 shows you how to call the `test_SimpleStruct()` function with derived type arguments (of `DerivedStruct_BaseStruct` type). Generally, this coding pattern can be used to pass either base type or derived type arguments.

Example 109: *Calling test_SimpleStruct() with Derived Type Arguments*

```
// C++
1  DerivedStruct_BaseStruct x;

   // Base members
2  x.setvarFloat((IT_Bus::Float) 3.14);
   x.setvarInt((IT_Bus::Int) 42);
   x.setvarString((IT_Bus::String) "BaseStruct-x");
   x.setvarAttrString((IT_Bus::String) "BaseStructAttr-x");
   // Derived members
   x.setvarFloatExt((IT_Bus::Float) -3.14f);
   x.setvarStringExt((IT_Bus::String) "DerivedStruct-x");
   x.setattrString1((IT_Bus::String) "DerivedAttr-x");

3  SimpleStruct x_holder(x);
4  SimpleStruct ret_holder;

5  proxy->test_SimpleStruct(x_holder, ret_holder);

6  const DerivedStruct_BaseStruct* ret_derived
       = dynamic_cast<const DerivedStruct_BaseStruct*>(
             ret_holder.get_derived()
         );

   // Use ret_derived type value...
   ...
```

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

1. The in parameter, `x`, of the `test_SimpleStruct()` function is declared to be of derived type, `DerivedStruct_BaseStruct`.
2. Both the base members and the derived members of the *in* parameter, `x`, are initialized here.
3. The derived type, `x`, is wrapped by a base type instance, `x_holder`. In this case, the `SimpleStruct` object, `x_holder`, is used purely as a holder type; `x_holder` does *not* directly represent a `SimpleStruct` type argument.

4. The return type, `ret_holder`, is declared to be of `SimpleStruct` type. Here also, `ret_holder` is treated as a holder type.
5. Call the remote `test_SimpleStruct()` function, passing in the two holder instances, `x_holder` and `ret_holder`.
6. To obtain a pointer to the derived type return value, call `SimpleStruct::get_derived()`. This function returns a pointer to the derived type contained in the `ret_holder` object. You can then cast the returned pointer to the appropriate type using the `dynamic_cast<>` operator.

   If necessary, you can call the `SimpleStruct::get_type()` function to discover the QName of the returned type before attempting to cast the return value.

Base type arguments only (for legacy code)

Example 110 shows you how to call the `test_SimpleStruct()` function with base type arguments (of `SimpleStruct` type). This coding pattern is supported for reasons of backward compatibility.

Example 110: *Calling test_SimpleStruct() with Base Type Arguments*

```
// C++
1  SimpleStruct x;

   // Base members
2  x.setvarFloat((IT_Bus::Float) 3.14);
   x.setvarInt((IT_Bus::Int) 42);
   x.setvarString((IT_Bus::String) "BaseStruct-x");
   x.setvarAttrString((IT_Bus::String) "BaseStructAttr-x");

3  SimpleStruct ret;

4  proxy->test_SimpleStruct(x, ret);

   // Use ret value...
   cout << ret.getvarFloat();
   ...
```

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

1. The in parameter, `x`, of the `test_SimpleStruct()` function is declared to be of base type, `SimpleStruct`.
2. The members of the `SimpleStruct` *in* parameter, `x`, are initialized.

3. The return value, `ret`, of the `test_SimpleStruct()` function is declared to be of base type, `SimpleStruct`.

   Note: The return value must be allocated *before* calling the `test_SimpleStruct()` function.

4. This line calls the remote `test_SimpleStruct()` function with in parameter, `x`, and return parameter, `ret`.

   Note: In this example, it is assumed that the return value is of base type, `SimpleStruct`. In general, however, the return type might be of derived type (see "Base or derived type arguments" on page 256).

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 263.

Limitations

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

- `all` complex types,
- `sequence` complex types.

Occurrence constraints are *not* supported within the scope of the following:

- `choice` complex types.

Element lists

Lists of elements appearing within a sequence complex type are represented in C++ by the `IT_Bus::ElementListT` template. You should not use this type directly in your code. Use the `IT_Vector` (see "IT_Vector Template Class" on page 307) in place of `IT_Bus::ElementListT`. The `IT_Bus::ElementListT` types automatically convert to and from `IT_Vector` types.

In addition to the standard member functions and operators defined by `IT_Vector`, the element list types support the following member functions:

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

size_t get_max_occurs() const;

void   set_size(size_t new_size);
```

```
size_t get_size() const;

const QName & get_item_name() const;
```

WSDL example

Example 111 shows the definition of a sequence type, SequenceType, which contains a list of integer elements followed by a list of string elements.

Example 111: *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 112 shows an outline of the C++ SequenceType class generated from Example 111 on page 260, which defines accessor and modifier functions for the varInt and varString elements.

Example 112: *Mapping of SequenceType to C++*

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

  SequenceType& operator= (const SequenceType& assign);

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

Example 112: *Mapping of SequenceType to C++*

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

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

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

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

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

  private:
  ...
};
```

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

C++ example

The following code fragment shows how to allocate and initialize an instance of `SequenceType` type containing two `varInt` elements and two `varString` elements:

```
// C++
SequenceType seq;

seq.getvarInt().set_size(2);
seq.getvarInt()[0] = 10;
seq.getvarInt()[1] = 20;
seq.getvarString().set_size(2);
seq.getvarString()[0] = "Zero";
seq.getvarString()[1] = "One";
```

Note how the `set_size()` function and `[]` operator are invoked directly on the member vectors, which are accessed by `getvarInt()` and `getvarString()` respectively. This is more efficient than creating a vector and passing it to `setvarInt()` or `setvarString()`, because it avoids creating unnecessary temporary vectors.

Alternatively, you could assign the member vectors, `seq.getvarInt()` and `seq.getvarString()`, to references of `IT_Vector` type and manipulate the references, `v1` and `v2`, instead. This is shown in the following code example:

```
// C++
SequenceType seq;

// Make a shallow copy of the vectors
IT_Vector<IT_Bus::Int>&    v1 = seq.getvarInt();
IT_Vector<IT_Bus::String>& v2 = seq.getvarString();

v1.push_back(10);
v1.push_back(20);
v2.push_back("Zero");
v2.push_back("One");
```

In this example, the vectors are initialized using the `push_back()` stack operation (adds an element to the end of the vector).

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

References

For more details about vector types see:

- The "IT_Vector Template Class" on page 307.
- 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 295.

Array definition syntax

An array is a sequence complex type that satisfies the following special conditions:

- The sequence complex type schema defines a *single* element only.
- The element definition has a `maxOccurs` attribute with a value greater than 1.

Note: All elements implicitly have `minOccurs=1` and `maxOccurs=1`, unless specified otherwise.

Hence, an Artix array definition has the following general syntax:

```
<complexType name="ArrayName">
    <sequence>
        <element name="ElemName" type="ElemType"
                 minOccurs="LowerBound" maxOccurs="UpperBound"/>
    </sequence>
</complexType>
```

The *ElemType* specifies the type of the array elements and the number of elements in the array can be anywhere in the range *LowerBound* to *UpperBound*.

Mapping to IT_Bus::ArrayT

When a sequence complex type declaration satisfies the special conditions to be an array, it is mapped to C++ differently from a regular sequence complex type. Instead of mapping to `IT_Bus::SequenceComplexType`, the array maps to the `IT_Bus::ArrayT<`*ElementType*`>` template type. Effectively, the C++ array template class can be treated like a vector.

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

WSDL array example

Example 113 shows how to define a one-dimensional string array, `ArrayOfString`, whose size can lie anywhere in the range `0` to unbounded.

Example 113: *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 114 shows how the `ArrayOfString` string array (from Example 113 on page 264) maps to C++.

Example 114: *Mapping of ArrayOfString to C++*

```
// C++
class ArrayOfString : public IT_Bus::ArrayT<IT_Bus::String>
{
  public:
    ArrayOfString();
    ArrayOfString(size_t dimension0);
    ArrayOfString(const ArrayOfString& copy);
    virtual ~ArrayOfString();

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

    ArrayOfString& operator= (const IT_Vector<IT_Bus::String>&
   assign);

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

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

Example 114: *Mapping of ArrayOfString to C++*

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

};

typedef IT_AutoPtr<ArrayOfString> ArrayOfStringPtr;
```

Notice that the C++ array class provides accessor functions, `getvarString()` and `setvarString()`, just like any other sequence complex type with occurrence constraints (see "Occurrence Constraints" on page 259). 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 115 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 311).

Example 115: *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 243 for a discussion of nested types). Example 116 shows an example of how to define a two-dimensional string array, `ArrayOfArrayOfString`.

Example 116: *Definition of a Multi-Dimensional String Array*

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

Example 116: *Definition of a Multi-Dimensional String Array*

```
            <complexType name="ArrayOfString">
                <sequence>
                    <element name="varString" type="xsd:string"
                             minOccurs="0" maxOccurs="unbounded"/>
                </sequence>
            </complexType>
            <complexType name="ArrayOfArrayOfString">
                <sequence>
                    <element name="nestArray"
                              type="xsd1:ArrayOfString"
                             minOccurs="0" maxOccurs="unbounded"/>
                </sequence>
            </complexType>
            ...
    ...
</definitions>
```

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

C++ example for multidimensional array

Example 117 shows an example of how to allocate and initialize a multi-dimensional array, of `ArrayOfArrayOfString` type.

Example 117: *C++ Example for a Multi-Dimensional Array*

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

for (int i = 0 ; i < a2.size(); i++) {
    a2[i].set_size(2);
}

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

The `set_size()` function enables you to set the dimension of each sub-array individually. If you choose different sizes for the sub-arrays, you can create `a2` as a ragged two-dimensional array.

Automatic conversion to IT_Vector

In general, a multi-dimensional array can automatically convert to a vector of `IT_Vector<`*SubArray*`>` type, where *SubArray* is the array element type.

Example 118 shows how an instance, `a2`, of `ArrayOfArrayOfString` type converts to an instance of `IT_Vector<ArrayOfString>` type by assignment.

Example 118: *Converting a Multi-Dimensional Array to IT_Vector Type*

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

for (int i = 0 ; i < a2.size(); i++) {
    a2[i].set_size(2);
}
...
// Obtain reference to the underlying IT_Vector type
IT_Vector<ArrayOfString>& v_a2 = a2;

cout << v_a2[0][0] << " " << v_a2[0][1] << " "
     << v_a2[1][0] << " " << v_a2[1][1] << endl;
cout << "v_a2.size() = " << v_a2.size() << endl;
```

References

For more details about vector types see:

- The "IT_Vector Template Class" on page 307.
- The section on C++ ANSI vectors in *The C++ Programming Language*, third edition, by Bjarne Stroustrup.

anyType Type

Overview

In an XML schema, the `xsd:anyType` is the base type from which other simple and complex types are derived. Hence, an element declared to be of `xsd:anyType` type can contain any XML type.

Note: The `xsd:anyType` is currently supported only by the CORBA, SOAP and XML bindings. Certain bindings—for example, Fixed, Tagged, TibMsg, and FML—do not support the use of `xsd:anyType` because they lack a corresponding construct.

Prerequisite for using anyType

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

anyType syntax

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

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

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

C++ mapping

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

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

- Inserting and extracting atomic types.
- Inserting and extracting user-defined types.

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

Inserting and extracting atomic types

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

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

For a complete list of the functions for the basic atomic types, see "AnyHolder API" on page 271.

For example, you can insert and extract an `xsd:short` integer to and from an `IT_Bus::AnyHolder` as follows:

```
// C++
// Insert an xsd:short value into an xsd:anyType.
IT_Bus::AnyHolder aH;
aH.set_short(1234);
...
// Extract an xsd:short value from an xsd:anyType.
IT_Bus::Short sh = aH.get_short();
```

Inserting and extracting user-defined types

To insert and extract user-defined types from an `IT_Bus::AnyHolder`, use the following functions:

```
void                    set_any_type(const IT_Bus::AnyType &);
IT_Bus::AnyType&        get_any_type();
const IT_Bus::AnyType&  get_any_type();
```

Note that all user-defined types inherit from `IT_Bus::AnyType`. There are no type-specific insertion or extraction functions generated for user-defined types.

Memory management for these functions is handled as follows:

- The `set_any_type()` function copies the inserted data.
- The `get_any_type()` functions do not copy the return value, rather they return either a writable (non-const) or read-only (const) reference to the data inside the `IT_Bus::AnyHolder`.

For example, given a user-defined sequence type, SequenceType (see the declaration in Example 86 on page 229), you can insert a SequenceType instance into an IT_Bus::AnyHolder as follows:

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

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

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

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

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

Accessing the type information

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

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

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

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

AnyHolder API

Example 119 shows the public API from the `IT_Bus::AnyHolder` class, including all of the function for inserting and extracting data values.

Example 119: *The IT_Bus::AnyHolder Class*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API AnyHolder : public AnyType
    {
      public:
        AnyHolder();
        virtual ~AnyHolder() ;
        ...
        virtual const QName & get_type() const ;
        ...
        //Set Methods
        void set_boolean(const IT_Bus::Boolean &);
        void set_byte(const IT_Bus::Byte &);
        void set_short(const IT_Bus::Short &);
        void set_int(const IT_Bus::Int &);
        void set_long(const IT_Bus::Long &);
        void set_string(const IT_Bus::String &);
        void set_float(const IT_Bus::Float &);
        void set_double(const IT_Bus::Double &);
        void set_ubyte(const IT_Bus::UByte &);
        void set_ushort(const IT_Bus::UShort &);
        void set_uint(const IT_Bus::UInt &);
        void set_ulong(const IT_Bus::ULong &);
        void set_decimal(const IT_Bus::Decimal &);

        void set_any_type(const AnyType&);

        //GET METHODS
        IT_Bus::Boolean & get_boolean();
        IT_Bus::Byte & get_byte();
        IT_Bus::Short & get_short();
        IT_Bus::Int & get_int();
        IT_Bus::Long & get_long();
        IT_Bus::String & get_string();
        IT_Bus::Float & get_float();
        IT_Bus::Double & get_double();
        IT_Bus::UByte & get_ubyte() ;
        IT_Bus::UShort & set_ushort();
        IT_Bus::UInt & get_uint();
        IT_Bus::ULong & set_ulong();
```

Example 119: *The IT_Bus::AnyHolder Class*

```
        IT_Bus::Decimal & get_decimal();

        AnyType& get_any_type();

        //CONST GET METHODS
        const IT_Bus::Boolean & get_boolean() const;
        const IT_Bus::Byte & get_byte() const;
        const IT_Bus::Short & get_short() const;
        const IT_Bus::Int & get_int() const;
        const IT_Bus::Long & get_long() const;
        const IT_Bus::String & get_string() const;
        const IT_Bus::Float & get_float() const;
        const IT_Bus::Double & get_double() const;
        const IT_Bus::UByte & get_ubyte() const;
        const IT_Bus::UShort & get_ushort() const;
        const IT_Bus::UInt & get_uint() const;
        const IT_Bus::ULong & get_ulong() const;
        const IT_Bus::Decimal & get_decimal() const;

        const AnyType& get_any_type() const;
        ...
    };
};
```

Nillable Types

Overview

This section describes how to define and use nillable types; that is, XML elements defined with `xsd:nillable="true"`.

In this section

This section contains the following subsections:

Introduction to Nillable Types

Overview

An element in an XML schema may be declared as nillable by setting the `nillable` attribute equal to `true`. This is useful in cases where you would like to have the option of transmitting no value for a type (for example, if you would like to define an operation with optional parameters).

Nillable syntax

To declare an element as nillable, use the following syntax:

`<element name="`*ElementName*`" type="`*ElementType*`" nillable="true"/>`

The `nillable="true"` setting indicates that this as a nillable element. If the `nillable` attribute is missing, the default is value is `false`.

On-the-wire format

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

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

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

C++ API for nillable types

Example 120 shows the public member functions of the `IT_Bus::NillableValueBase` class, which provides the C++ API for nillable types.

Example 120: *C++ API for Nillable Types*

```
// C++
namespace IT_Bus
{
    template <class T>
    class NillableValueBase : public Nillable
    {
      public:
        virtual ~NillableValueBase();
        virtual AnyType& operator=(const AnyType& other);

        virtual Boolean is_nil() const;
        virtual void set_nil();
        ...
        virtual const T&
```

Example 120: *C++ API for Nillable Types*

```
        get() const IT_THROW_DECL((NoDataException));

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

        // Set the data value, make is_nil() false.
        virtual void set(const T& data);

        // data != 0 ==> set the data value, make is_nil() false.
        // data == 0 ==> make is_nil() true.
        virtual void set(const T *data);

        // Reset to nil, makes is_nil() true.
        virtual void reset();

      protected:
    ...
};
```

Nillable Atomic Types

Overview

This subsection describes how to define and use XML schema nillable atomic types. In C++, every atomic type, *AtomicTypeName*, has a nillable counterpart, *AtomicTypeName*`Nillable`. For example, `IT_Bus::Short` has `IT_Bus::ShortNillable` as its nillable counterpart.

You can modify or access the value of an atomic nillable type, `T`, using the `T.set()` and `T.get()` member functions, respectively. For full details of the API for nillable types see "C++ API for nillable types" on page 274.

Table of nillable atomic types

Table 10 shows how the XML schema atomic types map to C++ when the `xsd:nillable` flag is set to `true`.

Table 10: *Nillable Atomic Types*

Schema Type	Nillable C++ Type
`xsd:anyType`	*Not supported as nillable*
`xsd:boolean`	`IT_Bus::BooleanNillable`
`xsd:byte`	`IT_Bus::ByteNillable`
`xsd:unsignedByte`	`IT_Bus::UByteNillable`
`xsd:short`	`IT_Bus::ShortNillable`
`xsd:unsignedShort`	`IT_Bus::UShortNillable`
`xsd:int`	`IT_Bus::IntNillable`
`xsd:unsignedInt`	`IT_Bus::UIntNillable`
`xsd:long`	`IT_Bus::LongNillable`
`xsd:unsignedLong`	`IT_Bus::ULongNillable`
`xsd:float`	`IT_Bus::FloatNillable`
`xsd:double`	`IT_Bus::DoubleNillable`
`xsd:string`	`IT_Bus::StringNillable`
`xsd:QName`	`IT_Bus::QNameNillable`

Table 10: *Nillable Atomic Types*

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

WSDL example

Example 121 defines four elements, test_string_x, test_short_y, test_int_return, and test_float_z, of nillable atomic type. This example shows how to use the nillable atomic types as the parameters of an operation, send_receive_nil_part.

Example 121: *WSDL Example Showing Some Nillable Atomic Types*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BaseService"
   targetNamespace="http://soapinterop.org/"
    ...
    xmlns:tns="http://soapinterop.org/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://soapinterop.org/xsd">
    <types>
        <schema targetNamespace="http://soapinterop.org/xsd"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
            ...
            <element name="test_string_x" nillable="true"
                     type="xsd:string"/>
            <element name="test_short_y" nillable="true"
                     type="xsd:short"/>
            <element name="test_int_return" nillable="true"
                     type="xsd:int"/>
            <element name="test_float_z" nillable="true"
                     type="xsd:float"/>
        </schema>
    </types>
    ...
    <message name="NilPartRequest">
        <part name="x" element="xsd1:test_string_x"/>
        <part name="y" element="xsd1:test_short_y"/>
```

Example 121: *WSDL Example Showing Some Nillable Atomic Types*

```
    </message>
    <message name="NilPartResponse">
        <part name="return" element="xsd1:test_int_return"/>
        <part name="y" element="xsd1:test_short_y"/>
        <part name="z" element="xsd1:test_float_z"/>
    </message>
    ...
    <portType name="BasePortType">
        <operation name="send_receive_nil_part">
            <input name="doclit_nil_part_request"
                                  message="tns:NilPartRequest"/>
            <output name="doclit_nil_part_response"
                                 message="tns:NilPartResponse"/>
        </operation>
    </portType>
    ...
```

C++ example

Example 122 shows how to use nillable atomic types, `IT_Bus::StringNillable`, `IT_Bus::ShortNillable`, `IT_Bus::IntNillable`, and `IT_Bus::FloatNillable`, in a simple C++ example.

Example 122: *Using Nillable Atomic Types as Operation Parameters*

```
// C++
IT_Bus::StringNillable  x("String for sending");
IT_Bus::ShortNillable  y(321);
IT_Bus::IntNillable  var_return;
IT_Bus::FloatNillable z;

try {
    // bc is a client proxy for the BasePortType port type.
    bc.send_receive_nil_part(x, y, var_return, z);
}
catch (IT_Bus::FaultException &ex) {
    // ... deal with the exception (not shown)
}

if (! y.is_nil()) { cout << "y = " << y.get() << endl; }
if (! z.is_nil()) { cout << "z = " << z.get() << endl; }

if (! var_return.is_nil()) {
    cout << "var_return = " << var_return.get() << endl;
}
```

The value of a nillable atomic type, `T`, can be initialized using either a constructor, `T()`, or the `T.set()` member function.

Before attempting to read the value of a nillable atomic type using `T.get()`, you should check that the value is non-nil using the `T.is_nil()` member function.

Nillable User-Defined Types

Overview

This subsection describes how to define and use nillable user-defined types. In C++, every user-defined type, *UserTypeName*, has a nillable counterpart, *UserTypeName*`Nillable`.

You can modify or access the value of a user-defined nillable type, `T`, using the `T.set()` and `T.get()` member functions, respectively. For full details of the API for nillable types see "C++ API for nillable types" on page 274.

WSDL example

Example 123 shows the definition of an XML schema `all` complex type, named `SOAPStruct`. This is a complex type with ordinary (that is, non-nillable) member elements.

Example 123: *WSDL Example of an All Complex Type*

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

C++ mapping

Example 124 shows how the SOAPStruct type maps to C++. In addition to the regular mapping, which produces the C++ SOAPStruct and SOAPStructPtr classes, the WSDL-to-C++ compiler also generates a nillable type, SOAPStructNillable, and an associated smart pointer type, SOAPStructNillablePtr.

Example 124: *C++ Mapping of the SOAPStruct All Complex Type*

```
// C++
namespace INTEROP
{
    class SOAPStruct : public IT_Bus::AllComplexType  { ... }
    typedef IT_AutoPtr<SOAPStruct> SOAPStructPtr;

    typedef IT_Bus::NillableValue<SOAPStruct>
                SOAPStructNillable;
    typedef IT_Bus::NillablePtr<SOAPStruct>
                SOAPStructNillablePtr;
};
```

The API for the SOAPStructNillable type is defined in "C++ API for nillable types" on page 274.

C++ example

The following C++ example shows how to initialize an instance of SOAPStructNillable type, s_nillable. The nillable type is created in two steps: first of all, a SOAPStruct instance, s, is initialized; then the SOAPStruct instance is used to initialize a SOAPStructNillable instance.

```
// C++
// Initialize a SOAPStruct instance.
INTEROP::SOAPStruct s;
s.setvarFloat(3.14);
s.setvarInt(1234);
s.setvarString("Hello world!");

// Initialize a SOAPStructNillable instance.
INTEROP::SOAPStructNillable s_nillable;
s_nillable.set(s);
```

The next C++ example shows how to access the contents of the `SOAPStructNillable` type. Note that before attempting to access the value of the `SOAPStructNillable` using `get()`, you should check that the value is not nil using `is_nil()`.

```
// C++
if (! s_nillable.is_nil()) {
    cout << "varFloat =  " << s_nillable.get().getvarFloat()
         << endl;
    cout << "varInt =    " << s_nillable.get().getvarInt()
         << endl;
    cout << "varString = " << s_nillable.get().getvarString()
         << endl;
}
```

Nested Atomic Type Nillable Elements

Overview

This subsection describes how to define and use complex types (except arrays) that have some nillable member elements. That is, the type as a whole is not nillable, although some of its elements are.

The WSDL-to-C++ compiler treats a type with nillable elements as a special case. If a member element, *ElementName*, is defined with `xsd:nillable` equal to `true`, the element's C++ modifiers and accessors are then primarily pointer based.

For example, given that a member element *ElementName* is of *AtomicType* type, the accessors and modifier would have the following signatures:

```
const AtomicType * getElementName() const;
AtomicType *       getElementName();
void               setElementName(const AtomicType * val);
```

And an additional convenience function that allows you to set an element value using pass-by-reference:

```
void               setElementName(const AtomicType & val);
```

Note: Arrays with nillable elements are treated differently—see "Nillable Elements of an Array" on page 292.

WSDL example

Example 125 defines a sequence complex type, `Nil_SOAPStruct`, which has some nillable elements, `varInt`, `varFloat`, and `varString`.

Example 125: *WSDL Example of a Sequence Type with Nillable Elements*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BaseService"
    targetNamespace="http://soapinterop.org/"
    ...
    xmlns:tns="http://soapinterop.org/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://soapinterop.org/xsd">
    <types>
        <schema targetNamespace="http://soapinterop.org/xsd"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
            ...
```

Example 125: *WSDL Example of a Sequence Type with Nillable Elements*

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

C++ mapping

Example 126 shows how the `Nil_SOAPStruct` sequence complex type is mapped to C++. Note how the accessors for the nillable member elements, get*ElementName*`()`, return a pointer instead of a value; and how the modifiers for the nillable member elements, set*ElementName*`()`, take either a pointer argument or a reference argument. For example, the `getvarInt()` function returns a pointer to an `IT_Bus::Int` rather an `IT_Bus::Int` value.

Example 126: *C++ Mapping of the Nil_SOAPStruct Sequence Type*

```
// C++
namespace INTEROP {
    class Nil_SOAPStruct : public IT_Bus::SequenceComplexType
    {
      public:
        Nil_SOAPStruct();
        Nil_SOAPStruct(const Nil_SOAPStruct& copy);
        virtual ~Nil_SOAPStruct();
        ...
        const IT_Bus::Int * getvarInt() const;
        IT_Bus::Int * getvarInt();
        void setvarInt(const IT_Bus::Int * val);
        void setvarInt(const IT_Bus::Int & val);

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

Example 126: *C++ Mapping of the Nil_SOAPStruct Sequence Type*

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

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

    typedef IT_AutoPtr<Nil_SOAPStruct> Nil_SOAPStructPtr;

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

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

C++ example

The following C++ example shows how to create and initialize a `Nil_SOAPStruct` instance. Notice, for example, how the `setvarInt(const IT_Bus::Int&)` convenience function allows you to pass the integer argument as a reference, `i`, instead of a pointer.

```
// C++
Nil_SOAPStruct nil_s;

IT_Bus::Float f = 3.14;
IT_Bus::Int   i = 1234;
IT_Bus::String s = "A non-nil string.";

nil_s.setvarInt(i);
nil_s.setvarFloat(f);
nil_s.setvarString(s);
```

The next C++ example shows how to read the nillable elements of the `Nil_SOAPStruct` instance. Note how the elements are checked for nilness by comparing the result of calling `get`*`ElementName`*`()` with `0`.

```
// C++
if (nil_s.getvarInt() != 0) {
    cout << "varInt = " << *nil_s.getvarInt() << endl;
}

if (nil_s.getvarFloat() != 0) {
    cout << "varFloat = " << *nil_s.getvarFloat() << endl;
}

if (nil_s.getvarString() != 0) {
    cout << "varString = " << *nil_s.getvarString() << endl;
}
```

Nested User-Defined Nillable Elements

Overview

This subsection describes how to define and use complex types that have nillable member elements of user-defined type.

The WSDL-to-C++ compiler treats user-defined nillable elements as a special case. As with nillable elements of atomic type, if a member element of user-defined type, *ElementName*, is defined with `xsd:nillable` equal to `true`, the element's C++ modifiers and accessors are then primarily pointer based.

For example, given that a member element *ElementName* is of *UserType* type, the accessors and modifier would have the following signatures:

```
const UserType * getElementName() const;
UserType *       getElementName();
void             setElementName(const UserType * val);
void             setElementName(const UserType & val);
```

Note: Arrays with nillable elements are treated differently—see "Nillable Elements of an Array" on page 292.

WSDL example

Example 127 defines a sequence complex type, `Nil_NestedSOAPStruct`, which includes a nillable element of `SOAPStruct` type, `varSOAP`.

Example 127: *WSDL Example of a Nillable All Type inside a Sequence Type*

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

Example 127: *WSDL Example of a Nillable All Type inside a Sequence Type*

```
                    <element name="varFloat" type="xsd:float"/>
                    <element name="varInt" type="xsd:int"/>
                   <element name="varString" type="xsd:string"/>
                </all>
            </complexType>
            ...
            <complexType name="Nil_NestedSOAPStruct">
                <sequence>
                    <element name="varInt" nillable="true"
                             type="xsd:int"/>
                    <element name="varSOAP" nillable="true"
                             type="xsd1:SOAPStruct"/>
                </sequence>
            </complexType>
            ...
        </schema>
    </types>
    ...
```

C++ mapping

Example 128 shows how the `Nil_NestedSOAPStruct` sequence complex type is mapped to C++. Note how the `getvarSOAP()` functions return a pointer to a `SOAPStruct` rather a `SOAPStruct` value.

Example 128: *C++ Mapping of the Nil_NestedSOAPStruct Type*

```
// C++
class Nil_NestedSOAPStruct : public IT_Bus::SequenceComplexType
{
  public:
    Nil_NestedSOAPStruct();
    Nil_NestedSOAPStruct(const Nil_NestedSOAPStruct& copy);
    virtual ~Nil_NestedSOAPStruct();
    ...
    const IT_Bus::Int * getvarInt() const;
    IT_Bus::Int *       getvarInt();
    void setvarInt(const IT_Bus::Int * val);
    void setvarInt(const IT_Bus::Int & val);

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

Example 128: *C++ Mapping of the Nil_NestedSOAPStruct Type*

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

NillablePtr types

To help you manage the memory associated with nillable elements of user-defined type, *UserType*, the WSDL-to-C++ utility generates a nillable smart pointer type, *UserType*`NillablePtr`. The `NillablePtr` template types are similar to the `std::auto_ptr<>` template types from the Standard Template Library—see "Smart Pointers" on page 46.

For example, the following extract from the generated *WSDLFileName*`_wsdlTypes.h` header file defines a `SOAPStructNillablePtr` type, which is used to represent `SOAPStruct` nillable pointers:

```
// C++
typedef IT_Bus::NillablePtr<SOAPStruct, &SOAPStructQName>
   SOAPStructNillablePtr;
```

Example 129 shows the API for the `NillablePtr` template class. A `NillablePtr` instance can be initialized using either a `NillablePtr()` constructor, a `set()` member function, or an `operator=()` assignment operator. The `is_nil()` member function tests the pointer for nilness.

Example 129: *The NillablePtr Template Class*

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

Example 129: *The NillablePtr Template Class*

```
        void set(const T* data);

        virtual Boolean is_nil() const;

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

C++ example

The following C++ example shows how to create and initialize a `Nil_NestedSOAPStruct` instance. Notice how the argument to `setvarSOAP()` is passed as a pointer, `&nillable_struct`.

```
// C++
// Construct a smart nillable pointer.
// The SOAPStruct memory is owned by the smart nillable pointer.
SOAPStruct nillable_struct;
nillable_struct.setvarFloat(3.14);
nillable_struct.setvarInt(4321);
nillable_struct.setvarString("Nillable struct element.");

// Construct a nested struct.
Nil_NestedSOAPStruct outer_struct;
IT_Bus::Int k = 4321
outer_struct.setvarInt(&k);

// MEMORY MANAGEMENT: The argument to setvarSOAP is deep copied.
outer_struct.setvarSOAP(&nillable_struct);
```

The next C++ example shows how to read the nillable elements of the `Nil_NestedSOAPStruct` instance. Note how the `varSOAP` element is checked for nilness by calling `is_nil()`.

```
// C++
IT_Bus::Int * int_p  = outer_struct.getvarInt();

// MEMORY MANAGEMENT: outer_struct owns the return value.
SOAPStruct *  nillable_struct_p = outer_struct.getvarSOAP();

if (int_p != 0) {
    cout << "varInt = " << *int_p << endl;
}

if (!nillable_struct_p.is_nil() ) {
    cout << "varSOAP = " << *nillable_struct_p << endl;
}
```

Nillable Elements of an Array

Overview

This subsection describes how to define and use array complex types with nillable array elements. To define an array with nillable elements, add a `nillable="true"` setting to the array element declaration.

An array with nillable elements has the following general syntax:

```
<complexType name="ArrayName">
    <sequence>
        <element name="ElemName" type="ElemType" nillable="true"
                 minOccurs="LowerBound" maxOccurs="UpperBound"/>
    </sequence>
</complexType>
```

The *ElemType* specifies the type of the array elements and the number of elements in the array can be anywhere in the range *LowerBound* to *UpperBound*.

WSDL example

Example 130 shows defines an array complex type, `Nil_SOAPArray` (the name indicates that the type is used in a SOAP example, not that it is defined using SOAP array syntax) which has nillable array elements, `item`.

Example 130: *WSDL Example of an Array with Nillable Elements*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BaseService"
   targetNamespace="http://soapinterop.org/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://soapinterop.org/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://soapinterop.org/xsd">
    <types>
        <schema targetNamespace="http://soapinterop.org/xsd"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
            ...
```

Example 130: *WSDL Example of an Array with Nillable Elements*

```
            <complexType name="Nil_SOAPArray">
                <sequence>
                    <element name="item" nillable="true"
                             type="xsd:short" minOccurs="10"
                             maxOccurs="10"/>
                </sequence>
            </complexType>
            ...
        </schema>
    </types>
    ...
```

C++ mapping

Example 131 shows how the `Nil_SOAPArray` array complex type is mapped to C++. Note that the array elements are of `IT_Bus::ShortNillable` type.

Example 131: *C++ Mapping of the Nil_SOAPArray Array Type*

```
// C++
namespace INTEROP {
    class Nil_SOAPArray
      : public IT_Bus::ArrayT<IT_Bus::ShortNillable,
   &Nil_SOAPArray_item_qname, 10, 10>
    {
      public:
        Nil_SOAPArray();
        Nil_SOAPArray(const Nil_SOAPArray& copy);
        Nil_SOAPArray(size_t dimensions[]);
        Nil_SOAPArray(size_t dimension0);
        virtual ~Nil_SOAPArray();

        ...
        const IT_Bus::ElementListT<IT_Bus::ShortNillable> &
        getitem() const;

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

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

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

Example 131: *C++ Mapping of the Nil_SOAPArray Array Type*

```
    typedef IT_AutoPtr<Nil_SOAPArray> Nil_SOAPArrayPtr;

    typedef IT_Bus::NillableValue<Nil_SOAPArray,
   &Nil_SOAPArrayQName> Nil_SOAPArrayNillable;

    typedef IT_Bus::NillablePtr<Nil_SOAPArray,
   &Nil_SOAPArrayQName> Nil_SOAPArrayNillablePtr;
};
```

C++ example

The following C++ example shows how to create and initialize a `Nil_SOAPArray` instance. Because each array element is of `IT_Bus::ShortNillable` type, the array elements must be initialized using the `set()` member function. Any elements not explicitly initialized are nil by default.

```
// C++
Nil_SOAPArray nil_s(10);
nil_s[0].set(10);
nil_s[1].set(20);
nil_s[2].set(30);
nil_s[3].set(40);
nil_s[4].set(50);
// The remaining five element values are left as nil.
```

The next C++ example shows how to access the nillable array elements. You should check each of the array elements for nilness using the `is_nil()` member function before attempting to read an array element value.

```
// C++
for (size_t i=0; i<10; i++) {
    if (! nil_s[i].is_nil()) {
        cout << "Nil_SOAPArray[" << i << "] = "
             << nil_s[i].get() << endl;
    }
}
```

SOAP Arrays

Overview

In addition to the basic array types described in "Arrays" on page 263, 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 132 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 132: *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 133 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 133: *Sample Encoding of ArrayOfSOAPString*

```
<ArrayOfSOAPString SOAP-ENC:arrayType="xsd:string[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 134 shows a C++ example of how to allocate and initialize an `ArrayOfSOAPString` instance with four elements.

Example 134: *C++ Example of Initializing an ArrayOfSOAPString Instance*

```
// C++
// Allocate SOAP array of String
const size_t extents[] = {4};
ArrayOfSOAPString a_str(extents);

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 300.
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 303). 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 135 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 135: *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 136 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 136: *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 137 shows a C++ example of how to allocate and initialize an `Array2OfInt` instance with dimensions, `[2,3]`.

Example 137: *Initializing an Array2OfInt SOAP Array*

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

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

Example 137: *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 138 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 138: *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 139 shows an example of how to initialize a sparse array of `Array2OfInt` type.

Example 139: *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 137 on page 301). 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 140 shows an example of how to read a sparse array of `Array2OfInt` type.

Example 140: *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 141 shows the encoding of a partially transmitted `ArrayOfSOAPString` instance.

Example 141: *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 11 are *not* defined in `IT_Vector`.

Table 11: *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 142 shows how to use these basic vector operations to initialize an integer vector with the first one hundred integer squares.

Example 142: *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 142 on page 309 can be written in a more idiomatic style using vector iterators, as shown in Example 143.

Example 143: *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 12 lists the member types defined in `IT_Vector<T>`.

Table 12: *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 13 lists the `IT_Vector` member functions returning iterators.

Table 13: *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 14 lists the `IT_Vector` element access operations.

Table 14: *Element Access Operations for IT_Vector<T>*

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

Stack operations

Table 15 lists the `IT_Vector` stack operations.

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

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

List operations

Table 16 lists the `IT_Vector` list operations.

Table 16: *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 17 lists the other operations supported by `IT_Vector`.

Table 17: *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 10

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 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 IONA's Orbix.

These alternative approaches are illustrated in Figure 28.

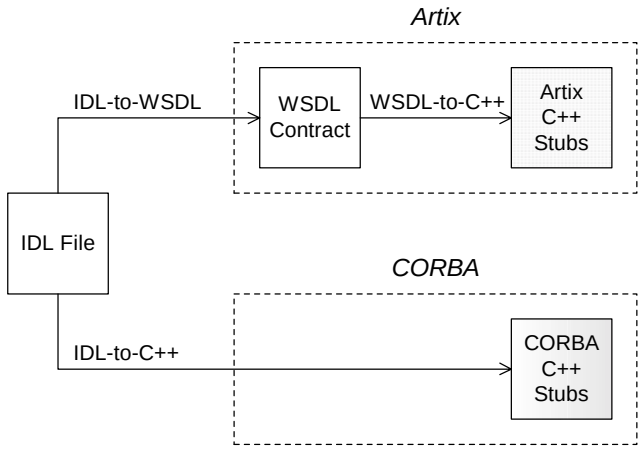

Figure 28: *Artix and CORBA Alternatives for IDL to C++ Mapping*

The advantage of using the Artix IDL-to-C++ mapping in an application is that it removes the CORBA dependency from your source code. For example, a server that implements an IDL interface using the Artix IDL-to-C++ mapping can also interoperate with other Web service protocols, such as SOAP over HTTP.

Unsupported IDL types

The following IDL types are not supported by the Artix C++ mapping:

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

IDL Basic Type Mapping

Overview

Table 18 shows how IDL basic types are mapped to WSDL and then to C++.

Table 18: *Artix Mapping of IDL Basic Types to C++*

IDL Type	WSDL Schema Type	C++ Type
`any`	`xsd:anyType`	`IT_Bus::AnyHolder`
`boolean`	`xsd:boolean`	`IT_Bus::Boolean`
`char`	`xsd:byte`	`IT_Bus::Byte`
`string`	`xsd:string`	`IT_Bus::String`
`wchar`	`xsd:string`	`IT_Bus::String`
`wstring`	`xsd:string`	`IT_Bus::String`
`short`	`xsd:short`	`IT_Bus::Short`
`long`	`xsd:int`	`IT_Bus::Int`
`long long`	`xsd:long`	`IT_Bus::Long`
`unsigned short`	`xsd:unsignedShort`	`IT_Bus::UShort`
`unsigned long`	`xsd:unsignedInt`	`IT_Bus::UInt`
`unsigned long long`	`xsd:unsignedLong`	`IT_Bus::ULong`
`float`	`xsd:float`	`IT_Bus::Float`
`double`	`xsd:double`	`IT_Bus::Double`
`long double`	*Not supported*	*Not supported*
`octet`	`xsd:unsignedByte`	`IT_Bus::UByte`
`fixed`	`xsd:decimal`	`IT_Bus::Decimal`
`Object`	`references:Reference`	`IT_Bus::Reference`

Mapping for string

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

IDL Complex Type Mapping

Overview

This section describes how the following IDL data types are mapped to WSDL and then to C++:

- enum type.
- struct type.
- union type.
- sequence types.
- array types.
- exception types.
- typedef of a simple type.
- typedef of a complex type.

enum type

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

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

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

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

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

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

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

Programming with the Enumeration Type

For details of how to use the enumeration type in C++, see "Deriving Simple Types by Restriction" on page 224.

---

union type

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

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

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

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

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

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

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

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

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

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

Programming with the Union Type

For details of how to use the union type in C++, see "Choice Complex Types" on page 232.

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 229.

---

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 263 and "IT_Vector Template Class" on page 307.

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

---

array types

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

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

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

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

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

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

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

Programming with Array Types

For details of how to use array types in C++, see "Arrays" on page 263 and "IT_Vector Template Class" on page 307.

---

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 33.

---

typedef of a simple type

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

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

The IDL-to-WSDL compiler maps the `SampleTypes::FloatAlias` typedef directory to the type, `xsd:float`.

The WSDL-to-C++ compiler then maps the `xsd:float` type directly to the `IT_Bus::Float` C++ type. Hence, no C++ typedef is generated for the `float` type.

---

typedef of a complex type

Consider the following IDL typedef that defines an alias of a `struct`, `SampleTypes::SampleStructAlias`:

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

The IDL-to-WSDL compiler maps the `SampleTypes::SampleStructAlias` typedef directly to the plain, unaliased `SampleTypes.SampleStruct` type.

The WSDL-to-C++ compiler then maps the `SampleTypes.SampleStruct` WSDL type directly to the `SampleTypes::SampleStruct` C++ type. Hence, no C++ typedef is generated for this struct type. Instead of a typedef, the C++ mapping uses the original, unaliased type.

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

IDL Module and Interface Mapping

Overview

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

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

Module mapping

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

Although IDL modules do *not* map to namespaces under the Artix C++ mapping, it is possible nevertheless to put generated C++ code into a namespace using the -n switch to the WSDL-to-C++ compiler (see "Generating Stub and Skeleton Code" on page 2). For example, if you pass a namespace, TEST, to the WSDL-to-C++ -n switch, the *ModuleName*::*Identifier* IDL identifier would map to TEST::*ModuleName_Identifier*.

Interface mapping

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

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

Object reference mapping

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

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

```
// IDL
interface Foo {};

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

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

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

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

See "Artix References" on page 75 for more details.

Operation mapping

Example 144 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 144: *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 144 on page 331, map to C++ as shown in Example 145,

Example 145: *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 146 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 146: *Example IDL Attributes*

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

The attributes from the preceding IDL, Example 146 on page 332, map to C++ as shown in Example 147,

Example 147:*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

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X