

Oracle® Service Architecture Leveraging Tuxedo (SALT)

Programming Guide

11g Release 1 (11.1.1.1.0)

March 2010

ORACLE®

Copyright © 2006, 2010, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on content, products and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Introduction to Oracle SALT Programming

Oracle SALT Web Services Programming	1-1
Oracle SALT Proxy Service	1-1
Oracle SALT Message Conversion	1-2
Oracle SALT Programming Tasks Quick Index	1-2
Oracle SALT SCA Programming	1-3

Data Type Mapping and Message Conversion

Overview of Data Type Mapping and Message Conversion	2-1
Understanding Oracle SALT Message Conversion	2-2
Inbound Message Conversion	2-2
Outbound Message Conversion	2-2
Tuxedo-to-XML Data Type Mapping for Tuxedo Services	2-3
Tuxedo STRING Typed Buffers	2-13
Tuxedo CARRAY Typed Buffers	2-14
Tuxedo MBSTRING Typed Buffers	2-16
Tuxedo XML Typed Buffers	2-17
Tuxedo VIEW/VIEW32 Typed Buffers	2-20
Tuxedo FML/FML32 Typed Buffers	2-22
Tuxedo X_C_TYPE Typed Buffers	2-26
Tuxedo X_COMMON Typed Buffers	2-27
Tuxedo X_OCTET Typed Buffers	2-27
Custom Typed Buffers	2-27
XML-to-Tuxedo Data Type Mapping for External Web Services	2-27
XML Schema Built-In Simple Data Type Mapping	2-28
XML Schema User Defined Data Type Mapping	2-32
WSDL Message Mapping	2-35

Web Service Client Programming

Overview	3-1
Oracle SALT Web Service Client Programming Tips	3-2
Web Service Client Programming References	3-7
Online References	3-7

Oracle Tuxedo ATMI Programming for Web Services

Overview	4-1
Converting WSDL Model Into Tuxedo Model	4-2
WSDL-to-Tuxedo Object Mapping	4-2
Invoking SALT Proxy Services	4-3
Oracle SALT Supported Communication Pattern	4-3
Tuxedo Outbound Call Programming: Main Steps	4-4
Managing Error Code Returned from GWWS	4-5
Handling Fault Messages in a Tuxedo Outbound Application	4-6

Using Oracle SALT Plug-Ins

Understanding Oracle SALT Plug-Ins	5-1
Plug-In Elements	5-1
Programming Message Conversion Plug-ins	5-7
How Message Conversion Plug-ins Work	5-7
When Do We Need Message Conversion Plug-in	5-10
Developing a Message Conversion Plug-in Instance	5-12
SALT 1.1 Custom Buffer Type Conversion Plug-in Compatibility	5-16
Programming Outbound Authentication Plug-Ins	5-17
How Outbound Authentication Plug-Ins Work	5-17
Implementing a Credential Mapping Interface Plug-In	5-18
Mapping the Tuxedo UID and HTTP Username	5-19

Oracle SALT SCA Programming

Overview	6-2
SCA Utilities	6-2
SCA Client Programming	6-2
SCA Client Programming Steps	6-3
SCA Component Programming	6-11
SCA Component Programming Steps	6-13
SCA Python and Ruby Programming	6-18
SCA Python and Ruby Programming Overview	6-19
Python and Ruby Client Programming	6-20
Python and Ruby Component Programming	6-21
SCA Structure Support	6-25
SCA Structure Support Overview	6-25
Using SCA Structure Description Files	6-27
Using tuxscagen to Generate Structures	6-29
SCA Remote Protocol Support	6-29
/WS	6-29
/Domains	6-30
SCA Binding	6-30
ATMI Binding	6-30
Java ATMI (JATMI) Binding	6-32
Python and Ruby Binding	6-35
Web Services Binding	6-37
SCA Data Type Mapping	6-41
Run-Time Data Type Mapping	6-42
SCA Utility Data Type Mapping	6-51
Python and Ruby Data Type Mapping	6-59

SCA Structure Data Type Mapping	6-66
SCA and Tuxedo Interoperability	6-69
SCA Transactions	6-69
SCA Security	6-70
See Also	6-70

Introduction to Oracle SALT Programming

This section includes the following topics:

- [Oracle SALT Web Services Programming](#)
- [Oracle SALT SCA Programming](#)

Oracle SALT Web Services Programming

Oracle SALT provides bi-directional connectivity between Tuxedo applications and Web service applications. Existing Tuxedo services can be easily exposed as Web Services without requiring additional programming tasks. Oracle SALT generates a WSDL file that describes the Tuxedo Web service contract so that any standard Web service client toolkit can be used to access Tuxedo services.

Web service applications (described using a WSDL document) can be imported as if they are standard Tuxedo services and invoked using Tuxedo ATMIs from various Tuxedo applications (for example, Tuxedo ATMI clients, ATMI servers, Jolt clients, COBOL clients, .NET wrapper clients and so on).

Oracle SALT Proxy Service

Oracle SALT proxy services are Tuxedo service entries advertised by the Oracle SALT Gateway, GWWS. The proxy services are converted from the Web service application WSDL file. Each WSDL file `wSDL:operation` object is mapped as one SALT proxy service.

The Oracle SALT proxy service is defined using the Service Metadata Repository service definition syntax. These service definitions must be loaded into the Service Metadata Repository. To invoke an proxy service from a Tuxedo application, you must refer to the Tuxedo Service Metadata Repository to get the service contract description.

For more information, see [?Sparatext>?](#).

Oracle SALT Message Conversion

To support Tuxedo application and Web service application integration, the Oracle SALT gateway converts SOAP messages into Tuxedo typed buffers, and vice versa. The message conversion between SOAP messages and Tuxedo typed buffers is subject to a set of SALT pre-defined basic data type mapping rules.

When exposing Tuxedo services as Web services, a set of Tuxedo-to-XML data type mapping rules are defined. The message conversion process conforms to Tuxedo-to-XML data type mapping rules is called “Inbound Message Conversion”.

When importing external Web services as SALT proxy services, a set of XML-to-Tuxedo data type mapping rules are defined. The message conversion process conforms to XML-to-Tuxedo data type mapping rules is called “Outbound Message Conversion”.

For more information about SALT message conversion and data type mapping, see [?Sparatext>?](#).

Oracle SALT Programming Tasks Quick Index

[Table 1-1](#) lists a quick index of Oracle SALT programming tasks. You can locate your programming tasks first and then click on the corresponding link for detailed description.

Table 1-1 Oracle SALT Programming Tasks Quick Index

	Tasks	Refer to ...
Invoking Tuxedo services (inbound) through Oracle SALT	Develop Web service client programs for Tuxedo services invocation	?Sparatext>? on page 3-2
	Understand inbound message conversion and data type mapping rules	?Sparatext>? on page 2-2 ?Sparatext>? on page 2-3
	Develop inbound message conversion plug-in	?Sparatext>? on page 5-7

Table 1-1 Oracle SALT Programming Tasks Quick Index

	Tasks	Refer to ...
Invoking external Web services (outbound) through Oracle SALT	Understand the general outbound service programming concepts	?Sparatext>? on page 4-1
	Understand outbound message conversion and data type mapping rules	?Sparatext>? on page 2-2 ?Sparatext>? on page 2-27
	Develop outbound message conversion plug-in	?Sparatext>? on page 5-7
	Develop your own plug-in to map Tuxedo user name with user name for outbound HTTP basic authentication	?Sparatext>? on page 5-17

Oracle SALT SCA Programming

SCA components run on top of the Oracle Tuxedo infrastructure using ATMI binding allowing you to better blend high-output, high-availability and scalable applications in your SOA environment. The Tuxedo SCA container is built on top of Tuscany SCA Native and Tuscany SDO C++ ((Assembly: 0.96, Client and Implementation Model 0.95) and SDO (2.01)).

The ATMI binding implementation provides native Tuxedo communications between SCA components as well as SCA components and Tuxedo programs (clients and servers). Runtime checks are encapsulated in an exception defined in a header (`tuxsca.h`) provided with the ATMI binding. This exception (`ATMIBindingException`), is derived from `ServiceRuntimeException` (so that programs not aware of the ATMI binding can still catch `ServiceRuntimeException`) and thrown back to the caller.

SCA deployment is handled by the following build commands:

- `buildscaclient`
- `buildscacomponent`
- `buildscaserver`

SCA clients can be stand-alone or part of a server, similar to Tuxedo ATMI clients. Components are first built using `buildscacomponent` and then Tuxedo-enabled using `buildscaserver`. SCA administration is performed using common Tuxedo commands (for example, `tadmin`), and the `scaadmin` command for SCA-specific tasks.

For more information, see:

Introduction to Oracle SALT Programming

- [Oracle SALT Administration Guide](#)
- [Oracle SALT Reference Guide](#)
- [SCA Service Component Architecture Client and Implementation Model Specification for C++](#)

Data Type Mapping and Message Conversion

This topic contains the following sections:

- [Overview of Data Type Mapping and Message Conversion](#)
- [Understanding Oracle SALT Message Conversion](#)
- [Tuxedo-to-XML Data Type Mapping for Tuxedo Services](#)
- [XML-to-Tuxedo Data Type Mapping for External Web Services](#)

Overview of Data Type Mapping and Message Conversion

Oracle SALT supports bi-directional data type mapping between WSDL messages and Tuxedo typed buffers. For each service invocation, GWWS server converts each message between Tuxedo typed buffer and SOAP message payload. SOAP message payload is the XML effective data encapsulated within the <soap:body> element. For more information, see [Understanding Oracle SALT Message Conversion](#).

For native Tuxedo services, each Tuxedo buffer type is described using an XML Schema in the SALT generated WSDL document. Tuxedo service request/response buffers are represented in regular XML format. For more information, see [Tuxedo-to-XML Data Type Mapping for Tuxedo Services](#).

For external Web services, each WSDL message is mapped as a Tuxedo FML32 buffer structure. A Tuxedo application invokes SALT proxy service using FML32 buffers as input/output. For more information see, [XML-to-Tuxedo Data Type Mapping for External Web Services](#).

Understanding Oracle SALT Message Conversion

Oracle SALT message conversion is the message transformation process between SOAP XML data and Tuxedo typed buffer. Oracle SALT introduces two types message conversion rules: Inbound Message Conversion and Outbound Message Conversion.

Inbound Message Conversion

Inbound message conversion process is the SOAP XML Payload and Tuxedo typed buffer conversion process conforms to the “Tuxedo-to-XML data type mapping rules”. Inbound message conversion process happens in the following two phases:

- When GWWS accepts SOAP requests for legacy Tuxedo services;
- When GWWS accepts response typed buffer from legacy Tuxedo service.

Oracle SALT encloses Tuxedo buffer content with element `<inbuf>`, `<outbuf>` and/or `<errbuf>` in the SOAP message, the content included within element `<inbuf>`, `<outbuf>` and/or `<errbuf>` is called “Inbound XML Payload”.

Outbound Message Conversion

Outbound message conversion process is the SOAP XML Payload and Tuxedo typed buffer conversion process conforms to the “Tuxedo-to-XML data type mapping rules”. Outbound message conferring process happens in the following two phases:

- When GWWS accepts request typed buffer sent from a Tuxedo application;
- When GWWS accepts SOAP response message from external Web service.

[Table 2-1](#) compares an inbound message conversion process and an outbound message conversion process.

Table 2-1 Inbound Message Conversion vs. Outbound Message Conversion

Inbound Message Conversion	Outbound Message Conversion
SOAP message payload is encapsulated with <code><inbuf></code> , <code><outbuf></code> or <code><errbuf></code>	SOAP message payload is the entire <code><soap:body></code>

Table 2-1 Inbound Message Conversion vs. Outbound Message Conversion

Inbound Message Conversion	Outbound Message Conversion
Transformation according to “Tuxedo-to-XML data type mapping rules”	Transformation according to “XML-to-Tuxedo data type mapping rules”
All Tuxedo buffer types are involved	Only Tuxedo FML32 buffer type is involved

Tuxedo-to-XML Data Type Mapping for Tuxedo Services

Oracle SALT provides a set of rules for describing Tuxedo typed buffers in an XML document as shown in [Table 2-2](#). These rules are exported as XML Schema definitions in SALT WSDL documents. This simplifies buffer conversion and does not require previous Tuxedo buffer type knowledge.

Data Type Mapping and Message Conversion

Table 2-2 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
STRING	Tuxedo STRING typed buffers are used to store character strings that terminate with a NULL character. Tuxedo STRING typed buffers are self-describing.	<p data-bbox="850 392 946 413">xsd:string</p> <p data-bbox="850 430 1233 543">In the SOAP message, the XML element that encapsulates the actual string data, must be defined using <code>xsd:string</code> directly.</p> <p data-bbox="850 560 915 581">Notes:</p> <ul data-bbox="850 598 1233 1097" style="list-style-type: none"> <li data-bbox="850 598 1233 972">• The STRING data type can be specified with a max data length in the Tuxedo Service Metadata Repository. If defined in Tuxedo, the corresponding SOAP message also enforces this maximum. The GWWS server validates the actual message byte length against the definition in Tuxedo Service Metadata Repository. A SOAP fault message is returned if the message byte length exceeds supported maximums. <li data-bbox="850 989 1233 1097">• If GWWS server receives a SOAP message other than “UTF-8”, the corresponding string value is in the same encoding.

Table 2-2 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
CARRAY (Mapping with SOAP Message plus Attachments)	Tuxedo CARRAY typed buffers store character arrays, any of which can be NULL. CARRAY buffers are used to handle data opaquely and are not self-describing.	<p>The CARRAY buffer raw data is carried within a MIME multipart/related message, which is defined in the “SOAP Messages with Attachments” specification.</p> <p>The two data formats supported for MIME Content-Type attachments are:</p> <ul style="list-style-type: none"> • application/octet-stream <ul style="list-style-type: none"> – For Apache Axis • text/xml <ul style="list-style-type: none"> – For Oracle WebLogic Server <p>The format depends on which Web service client-side toolkit is used.</p> <p>Note: The SOAP with Attachment rule is only interoperable with Oracle WebLogic Server and Apache Axis.</p> <p>Note: CARRAY data types can be specified with a max byte length. If defined in Tuxedo, the corresponding SOAP message is enforced with this limitation. The GWWS server validates the actual message byte length against the definition in the Tuxedo Service Metadata Repository.</p>

Table 2-2 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
CARRAY (Mapping with base64Binary)	Tuxedo CARRAY typed buffers store character arrays, any of which can be NULL. CARRAY buffers are used to handle data opaquely and are not self-describing.	<p data-bbox="848 392 1072 418"><code>xsd:base64Binary</code></p> <p data-bbox="848 430 1233 630">The CARRAY data bytes must be encoded with <code>base64Binary</code> before it can be embedded in a SOAP message. Using <code>base64Binary</code> encoding with this opaque data stream saves the original data and makes the embedded data well-formed and readable.</p> <p data-bbox="848 647 1233 760">In the SOAP message, the XML element that encapsulates the actual CARRAY data, must be defined with <code>xsd:base64Binary</code> directly.</p> <p data-bbox="848 782 1233 1072">Note: CARRAY data type can be specified with a max byte length. If defined in Tuxedo, the corresponding SOAP message is enforced with this limitation. The GWWS server validates the actual message byte length against the definition in the Tuxedo Service Metadata Repository.</p>

Table 2-2 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
MBSTRING	<p>Tuxedo MBSTRING typed buffers are used for multibyte character arrays. Tuxedo MBSTRING buffers consist of the following three elements:</p> <ul style="list-style-type: none"> • Code-set character encoding • Data length • Character array of the encoding. 	<p><code>xsd:string</code></p> <p>The XML Schema built-in type, <code>xsd:string</code>, represents the corresponding type for buffer data stored in a SOAP message.</p> <p>The GWWS server only accepts “UTF-8” encoded XML documents. If the Web service client wants to access Tuxedo services with MBSTRING buffer, the mbstring payload must be represented as “UTF-8” encoding in the SOAP request message.</p> <p>Note: The GWWS server transparently passes the “UTF-8” character set string to the Tuxedo service using MBSTRING Typed buffer format. The actual Tuxedo services handles the UTF-8 string.</p> <p>For any Tuxedo response MBSTRING typed buffer (with any encoding character set), The GWWS server automatically transforms the string into “UTF-8” encoding and sends it back to the Web service client.</p>
MBSTRING (cont.)		<p>Limitation:</p> <p>Tuxedo MBSTRING data type can be specified with a max byte length in the Tuxedo Service Metadata Repository. The GWWS server checks the byte length of the converted MBSTRING buffer value.</p> <p>Note: Max byte length value is not used to enforce the character number contained in the SOAP message.</p>

Table 2-2 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
XML	Tuxedo XML typed buffers store XML documents.	<p data-bbox="848 392 1002 413"><code>xsd:anyType</code></p> <p data-bbox="848 430 1233 600">The XML Schema built-in type, <code>xsd:anyType</code>, is the corresponding type for XML documents stored in a SOAP message. It allows you to encapsulate any well-formed XML data within the SOAP message.</p> <p data-bbox="848 618 964 638">Limitation:</p> <p data-bbox="848 656 1233 767">The GWWS server validates that the actual XML data is well-formed. It will not do any other enforcement validation, such as Schema validation.</p> <p data-bbox="848 784 1233 868">Only a single root XML buffer is allowed to be stored in the SOAP body; the GWWS server checks for this.</p> <p data-bbox="848 885 1233 1027">The actual XML data must be encoded using the “UTF-8” character set. Any original XML document prolog information cannot be carried within the SOAP message.</p> <p data-bbox="848 1045 1233 1156">XML data type can specify a max byte data length. If defined in Tuxedo, the corresponding SOAP message must also enforce this limitation.</p> <p data-bbox="848 1173 1233 1437">Note: The Oracle SALT WSDL generator will not have <code>xsd:maxLength</code> restrictions in the generated WSDL document, but the GWWS server will validate the byte length according to the Tuxedo Service Metadata Repository definition.</p>
X_C_TYPE	X_C_TYPE buffer types are equivalent to VIEW buffer types.	See VIEW/VIEW32

Table 2-2 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
X_COMMON	X_COMMON buffer types are equivalent to VIEW buffer types, but are used for compatibility between COBOL and C programs. Field types should be limited to short, long, and string	See VIEW/VIEW32
X_OCTET	X_OCTET buffer types are equivalent to CARRAY buffer types	See CARRAY <code>xsd:base64Binary</code>

Table 2-2 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
VIEW/VIEW32	<p>Tuxedo VIEW and VIEW32 typed buffers store C structures defined by Tuxedo applications.</p> <p>VIEW structures are defined by using VIEW definition files. A VIEW buffer type can define multiple fields.</p> <p>VIEW supports the following field types:</p> <ul style="list-style-type: none"> • short • int • long • float • double • char • string • carray <p>VIEW32 supports all the VIEW field types and mbstring.</p>	<p>Each VIEW or VIEW32 data type is defined as an XML Schema complex type. Each VIEW field should be one or more sub-elements of the XML Schema complex type. The name of the sub-element is the VIEW field name. The occurrence of the sub-element depends on the count attribute of the VIEW field definition. The value of the sub-element should be in the VIEW field data type corresponding XML Schema type.</p> <p>The the field types and the corresponding XML Schema type are listed as follows:</p> <ul style="list-style-type: none"> • short maps to xsd:short • int maps to xsd:int • long maps to xsd:long • float maps to xsd:float • double maps to xsd:double • char (defined as byte in Tuxedo Service Metadata Repository definition) maps to xsd:byte • char (defined as char in Tuxedo Service Metadata Repository definition) maps to xsd:string (with restrictions maxlength=1) • string maps to xsd:string • carray maps to xsd:base64Binary • mbstring maps to xsd:string
VIEW/VIEW32 (cont.)		<p>For more information, see VIEW/VIEW32 Considerations on page 2?21.</p>

Table 2-2 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
FML/FML32	<p>Tuxedo FML and FML32 type buffers are proprietary Oracle Tuxedo system self-describing buffers. Each data field carries its own identifier, an occurrence number, and possibly a length indicator.</p> <p>FML supports the following field types:</p> <ul style="list-style-type: none"> • FLD_CHAR • FLD_SHORT • FLD_LONG • FLD_FLOAT • FLD_DOUBLE • FLD_STRING • FLD_CARRAY <p>FML32 supports all the FML field types and FLD_PTR, FLD_MBSTRING, FLD_FML32, and FLD_VIEW32.</p>	<p>FML/FML32 buffers can only have basic data-dictionary-like definitions for each basic field data. A particular FML/FML32 buffer definition should be applied for each FML/FML32 buffer with a different type name.</p> <p>Each FML/FML32 field should be one or more sub-elements within the FML/FML32 buffer XML Schema type. The name of the sub-element is the FML field name. The occurrence of the sub-element depends on the count and required count attribute of the FML/FML32 field definition.</p> <p>The field types and the corresponding XML Schema type are listed below:</p> <ul style="list-style-type: none"> • short maps to xsd:short • int maps to xsd:int • long maps to xsd:long • float maps to xsd:float • double maps to xsd:double • char (defined as byte in Tuxedo Service Metadata Repository definition) maps to xsd:byte • char (defined as char in Tuxedo Service Metadata Repository definition) maps to xsd:string • string maps to xsd:string • carray maps to xsd:base64Binary • mbstring maps to xsd:string

Table 2-2 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
FML/FML32 (cont.)		<ul style="list-style-type: none"> <li data-bbox="848 392 1233 444">• view32 maps to <code>tuxtype:view</code> <code><viewname></code> <li data-bbox="848 458 1233 510">• fml32 maps to <code>tuxtype:fml32</code> <code><svcname>_p<SeqNum></code> <p data-bbox="848 524 1233 661">To avoid multiple embedded FML32 buffers in an FML32 buffer, a unique sequence number (<code><SeqNum></code>) is used to distinguish the embedded FML32 buffers.</p> <p data-bbox="848 692 1233 840">Note: <code>ptr</code> is not supported. For limitations and considerations regarding mapping FML/FML32 buffers, refer to FML/FML32 Considerations on page 2?25.</p>

Tuxedo STRING Typed Buffers

Tuxedo STRING typed buffers are used to store character strings that end with a NULL character. Tuxedo STRING typed buffers are self-describing.

[Listing 2-1](#) shows a SOAP message for a `TOUPPER` Tuxedo service example that accepts a STRING typed buffer.

Listing 2-1 Soap Message for a String Typed Buffer in TOUPPER Service

```
<?xml ... encoding="UTF-8" ?>
.....
<SOAP:body>
  <m:TOUPPER xmlns:m="urn:.....">
    <inbuf>abcdefg</inbuf>
  </m:TOUPPER>
</SOAP:body>
```

The XML Schema for `<inbuf>` is:

```
<xsd:element name="inbuf" type="xsd:string" />
```

Tuxedo CARRAY Typed Buffers

Tuxedo CARRAY typed buffers are used to store character arrays, any of which can be NULL. They are used to handle data opaquely and are not self-describing. Tuxedo CARRAY typed buffers can map to `xsd:base64Binary` or MIME attachments. The default is `xsd:base64Binary`.

Mapping Example Using `base64Binary`

[Listing 2-2](#) shows the SOAP message for the `TOUPPER` Tuxedo service, which accepts a CARRAY typed buffer using `base64Binary` mapping.

Listing 2-2 Soap Message for a CARRAY Typed Buffer Using `base64Binary` Mapping

```
<SOAP:body>
  <m:TOUPPER xmlns:m="urn:.....">
    <inbuf>QWxhZGRpbjpvYVUuIHNLc2FtZQ==</inbuf>
  </m:TOUPPER>
</SOAP:body>
```

The XML Schema for `<inbuf>` is:

```
<xsd:element name="inbuf" type="xsd:base64Binary" />
```

Mapping Example Using MIME Attachment

[Listing 2-3](#) shows the SOAP message for the `TOUPPER` Tuxedo service, which accepts a CARRAY typed buffer as a MIME attachment.

Listing 2-3 Soap Message for a CARRAY Typed Buffer Using MIME Attachment

```
MIME-Version: 1.0
Content-Type: Multipart/Related; boundary=MIME_boundary; type=text/xml;
  start="<claim061400a.xml@example.com>"
```

Content-Description: This is the optional message description.

```
--MIME_boundary
```

```
Content-Type: text/xml; charset=UTF-8
```

```
Content-Transfer-Encoding: 8bit
```

```
Content-ID: <claim061400a.xml@ example.com>
```

```
<?xml version='1.0' ?>
```

```
<SOAP-ENV:Envelope
```

```
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
```

```
<SOAP-ENV:Body>
```

```
..
```

```
<m:TOUPPER xmlns:m="urn:...">
```

```
<inbuf href="cid:claim061400a.carray@example.com" />
```

```
</m:TOUPPER>
```

```
..
```

```
</SOAP-ENV:Body>
```

```
</SOAP-ENV:Envelope>
```

```
--MIME_boundary
```

```
Content-Type: text/xml
```

```
Content-Transfer-Encoding: binary
```

```
Content-ID: <claim061400a.carray @example.com>
```

```
...binary carray data...
```

```
--MIME_boundary--
```

The WSDL for carray typed buffer will look like the following:

```
<wsdl:definitions ...>
```

```
<wsdl:types ...>
```

```
  <xsd:schema ...>
```

```
    <xsd:element name="inbuf" type="xsd:base64Binary" />
```

```
  </xsd:schema>
```

```
</wsdl:types>
```

```
.....
```

```
<wsdl:binding ...>
```

```
  <wsdl:operation name="TOUPPER">
```

```
<soap:operation ...>
  <input>
    <mime:multipartRelated>
      <mime:part>
        <soap:body parts="..." use="..." />
      </mime:part>
      <mime:part>
        <mime:content part="..." type="text/xml"/>
      </mime:part>
    </mime:multipartRelated>
  </input>
  .....
</wsdl:operation>
</wsdl:binding>

</wsdl:definitions>
```

Tuxedo MBSTRING Typed Buffers

Tuxedo MBSTRING typed buffers are used for multibyte character arrays. Tuxedo MBSTRING typed buffers consist of the following three elements:

- code-set character encoding
- data length
- character array encoding.

Note: You cannot embed multibyte characters with non “UTF-8” code sets in the SOAP message directly.

[Listing 2-4](#) shows the SOAP message for the `MBSERVICE` Tuxedo service, which accepts an MBSTRING typed buffer.

Listing 2-4 SOAP Message for an MBSIRING Buffer

```
<?xml encoding="UTF-8"?>
  <SOAP:body>
```

```
<m:MBSERVICE xmlns:m="http://.....">
  <inbuf> こんにちは </inbuf>
</m:MBSERVICE>
```

The XML Schema for <inbuf> is:

```
<xsd:element name="inbuf" type="xsd:string" />
```

WARNING: Oracle SALT converts the Japanese character "—" (EUC-JP 0xa1bd, Shift-JIS 0x815c) into UTF-16 0x2015.

If you use another character set conversion engine, the EUC-JP or Shift-JIS multibyte output for this character may be different. For example, the Java i18n character conversion engine, converts this symbol to UTF-16 0x2014. The result is the also same when converting to UTF-8, which is the Oracle SALT default.

If you use another character conversion engine and Japanese "—" is included in MBSTRING, TUXEDO server-side MBSTRING auto-conversion cannot convert it back into Shift-JIS or EUC-JP.

Tuxedo XML Typed Buffers

Tuxedo XML typed buffers store XML documents.

[Listing 2-5](#) shows the Stock Quote XML document.

[Listing 2-6](#) shows the SOAP message for the STOCKING Tuxedo service, which accepts an XML typed buffer.

Listing 2-5 Stock Quote XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- "Stock Quotes". -->
<stockquotes>
  <stock_quote>
    <symbol>BEAS</symbol>
    <when>
      <date>01/27/2001</date>
      <time>3:40PM</time>
```

```
</when>
<change>+2.1875</change>
<volume>7050200</volume>
</stock_quote>
</stockquotes>
```

Then part of the SOAP message will look like the following:

Listing 2-6 SOAP Message for an XML Buffer

```
<SOAP:body>
  <m: STOCKINQ xmlns:m="urn:.....">
    <inbuf>
      <stockquotes>
        <stock_quote>
          <symbol>BEAS</symbol>
          <when>
            <date>01/27/2001</date>
            <time>3:40PM</time>
          </when>
          <change>+2.1875</change>
          <volume>7050200</volume>
        </stock_quote>
      </stockquotes>
    </inbuf>
  </m: STOCKINQ >
</SOAP:body>
```

The XML Schema for <inbuf> is:

```
<xsd:element name="inbuf" type="xsd:anyType" />
```

Note: If a default namespace is contained in a Tuxedo XML typed buffer and returned to the GWWS server, the GWWS server converts the default namespace to a regular name. Each element is then prefixed with this name.

For example, if a Tuxedo service returns a buffer having a default namespace to the GWWS server as shown in [Listing 2-7](#), the GWWS server converts the default namespace to a regular name as shown in [Listing 2-8](#).

Listing 2-7 Default Namespace Before Sending to GWWS Server

```
<Configuration xmlns="http://www.bea.com/Tuxedo/Salt/200606">
  <Servicelist id="simpapp">
    <Service name="toupper"/>
  </Servicelist>
  <Policy/>
  <System/>
  <WSGateway>
    <GWInstance id="GWWS1">
      <HTTP address="//myhost:8080"/>
    </GWInstance>
  </WSGateway>
</Configuration>
```

Listing 2-8 GWWS Server Converts Default Namespace to Regular Name

```
<dom0:Configuration
xmlns:dom0="http://www.bea.com/Tuxedo/Salt/200606">
  <dom0:Servicelist dom0:id="simpapp">
    <dom0:Service dom0:name="toupper"/>
  </dom0:Servicelist>
  <dom0:Policy></dom0:Policy>
  <dom0:System></dom0:System>
  <dom0:WSGateway>
    <dom0:GWInstance dom0:id="GWWS1">
      <dom0:HTTP dom0:address="//myhost:8080"/>
    </dom0:GWInstance>
  </dom0:WSGateway>
</dom0:Configuration>
```

Tuxedo VIEW/VIEW32 Typed Buffers

Tuxedo VIEW and VIEW32 typed buffers are used to store C structures defined by Tuxedo applications. You must define the VIEW structure with the VIEW definition files. A VIEW buffer type can define multiple fields.

[Listing 2-9](#) shows the MYVIEW VIEW definition file.

[Listing 2-10](#) shows the SOAP message for the MYVIEW Tuxedo service, which accepts a VIEW typed buffer.

Listing 2-9 VIEW Definition File for MYVIEW Service

```
VIEW MYVIEW
#type      cname      fbname      count      flag      size      null
float      float1     -           1          -         -         0.0
double     double1    -           1          -         -         0.0
long       long1      -           3          -         -         0
string     string1    -           2          -         20        '\0'
END
```

Listing 2-10 SOAP Message for a VIEW Typed Buffer

```
<SOAP:body>
  <m: STOCKINQ xmlns:m="http://.....">
    <inbuf>
      <float1>12.5633</float1>
      <double1>1.3522E+5</double1>
      <long1>1000</long1>
      <long1>2000</long1>
      <long1>3000</long1>
      <string1>abcd</string1>
      <string1>ubook</string1>
    </inbuf>
  </m: STOCKINQ >
</SOAP:body>
```

The XML Schema for <inbuf> is shown in [Listing 2-11](#).

Listing 2-11 XML Schema for a VIEW Typed Buffer

```
<xsd:complexType name=" view_MYVIEW">
  <xsd:sequence>
    <xsd:element name="float1" type="xsd:float" />
    <xsd:xsd:element name="double1" type="xsd:double" />
    <xsd:element name="long1" type="xsd:long" minOccurs="3" />
    <xsd:element name="string1" type="xsd:string minOccurs="3" />
  </xsd:sequence>
</xsd: complexType >
<xsd:element name="inbuf" type="tu xtype:view_MYVIEW" />
```

VIEW/VIEW32 Considerations

The following considerations apply when converting Tuxedo VIEW/VIEW32 buffers to and from XML.

- You must create an environment for converting XML to and from VIEW/VIEW32. This includes setting up a VIEW directory and system VIEW definition files. These definitions are automatically loaded by the GWWS server.
- The GWWS server provides strong consistency checking between the Tuxedo Service Metadata Repository VIEW/VIEW32 parameter definition and the VIEW/VIEW32 definition file at start up.

If an inconsistency is found, the GWWS server cannot start. Inconsistency messages are printed in the ULOG file.

- `tmwsdlgen` also provides strong consistency checking between the Tuxedo Service Metadata Repository VIEW/VIEW32 parameter definition and the VIEW/VIEW32 definition file at start up. If an inconsistency is found, the GWWS server will not start. Inconsistency messages are printed in the ULOG file.

If the VIEW definition file cannot be loaded, `tmwsdlgen` attempts to use the Tuxedo Service Metadata Repository definitions to compose the WSDL document.

- Because `dec_t` is not supported, if you define VIEW fields with type `dec_t`, the service cannot be exported as a Web service and an error message is generated when the Oracle SALT configuration file is loading.
- Although the Tuxedo Service Metadata Repository may define a size attribute for “string/mbstring” typed parameters (which represents the maximum byte length that is allowed in the Tuxedo typed buffer), Oracle SALT does not expose such restriction in the generated WSDL document.
- When a VIEW32 embedded MBString buffer is requested and returned to the GWWS server, the GWWS miscalculates the required MBString length and reports that the input string exceeds the VIEW32 maxlength. This is because the header is included in the transfer encoding information. You must include the header size when defining the VIEW32 field length.
- The Tuxedo primary data type “long” is indefinite between 32-bit and 64-bit scope, depending on the platform. However, the corresponding `xsd:long` schema type is used to describe 64-bit numeric values.

If the GWWS server runs in 32-bit mode, and the Web service client sends `xsd:long` typed data that exceeds the 32-bit value range, you may get a SOAP fault.

Tuxedo FML/FML32 Typed Buffers

Tuxedo FML and FML32 typed buffer are proprietary Oracle Tuxedo system self-describing buffers. Each data field carries its own identifier, an occurrence number, and possibly a length indicator.

FML Data Mapping Example

[Listing 2-12](#) shows the SOAP message for the `TRANSFER` Tuxedo service, which accepts an FML typed buffer.

The request fields for service `LOGIN` are:

```
ACCOUNT_ID      1      long           /* 2 occurrences, The withdrawal
account is 1st, and the deposit account is 2nd */
AMOUNT          2      float          /* The amount to transfer */
```

Part of the SOAP message is as follows:

Listing 2-12 SOAP Message for an FML Typed Buffer

```

<SOAP:body>
  <m:TRANSFER xmlns:m="urn:.....">
    <inbuf>
      <ACCOUNT_ID>40069901</ACCOUNT_ID>
      <ACCOUNT_ID>40069901</ACCOUNT_ID>
      <AMOUNT>200.15</AMOUNT>
    </inbuf>
  </m:TRANSFER >
</SOAP:body>

```

The XML Schema for <inbuf> is shown in [Listing 2-13](#).

Listing 2-13 XML Schema for an FML Typed Buffer

```

<xsd:complexType name=" fml_TRANSFER_In">
  <xsd:sequence>
    <xsd:element name="ACCOUNT_ID" type="xsd:long" minOccurs="2"/>
    <xsd:element name=" AMOUNT" type="xsd:float" />
  </xsd:sequence>
</xsd: complexType >
<xsd:element name="inbuf" type="tuatype: fml_TRANSFER_In" />

```

FML32 Data Mapping Example

[Listing 2-14](#) shows the SOAP message for the TRANSFER Tuxedo service, which accepts an FML32 typed buffer.

The request fields for service LOGIN are:

```

CUST_INFO          1          fml32          /* 2 occurrences, The withdrawal
customer is 1st, and the deposit customer is 2nd */
ACCOUNT_INFO      2          fml32          /* 2 occurrences, The withdrawal
account is 1st, and the deposit account is 2nd */
AMOUNT            3          float          /* The amount to transfer */

```

Each embedded CUST_INFO includes the following fields:

CUST_NAME	10	string
CUST_ADDRESS	11	carray
CUST_PHONE	12	long

Each embedded ACCOUNT_INFO includes the following fields:

ACCOUNT_ID	20	long
ACCOUNT_PW	21	carray

Part of the SOAP message will look as follows:

Listing 2-14 SOAP Message for Service with FML32 Buffer

```
<SOAP:body>
  <m:STOCKING xmlns:m="urn:.....">
    <inbuf>
      <CUST_INFO>
        <CUST_NAME>John</CUST_NAME>
        <CUST_ADDRESS>Building 15</CUST_ADDRESS>
        <CUST_PHONE>1321</CUST_PHONE>
      </CUST_INFO>
      <CUST_INFO>
        <CUST_NAME>Tom</CUST_NAME>
        <CUST_ADDRESS>Building 11</CUST_ADDRESS>
        <CUST_PHONE>1521</CUST_PHONE>
      </CUST_INFO>
      <ACCOUNT_INFO>
        <ACCOUNT_ID>40069901</ACCOUNT_ID>
        <ACCOUNT_PW>abc</ACCOUNT_PW>
      </ACCOUNT_INFO>
      <ACCOUNT_INFO>
        <ACCOUNT_ID>40069901</ACCOUNT_ID>
        <ACCOUNT_PW>zyx</ACCOUNT_PW>
      </ACCOUNT_INFO>

      <AMOUNT>200.15</AMOUNT>
    </inbuf>
```

```

    </m: STOCKINQ >
</SOAP:body>

```

The XML Schema for <inbuf> is shown in [Listing 2-15](#).

Listing 2-15 XML Schema for an FML32 Buffer

```

<xsd:complexType name="fml32_TRANSFER_In">
  <xsd:sequence>
    <xsd:element name="CUST_INFO" type="tuctype:fml32_TRANSFER_p1"
minOccurs="2" />
    <xsd:element name="ACCOUNT_INFO" type="tuctype:fml32_TRANSFER_p2"
minOccurs="2" />
    <xsd:element name="AMOUNT" type="xsd:float" />
  /xsd:sequence>
</xsd:complexType >

<xsd:complexType name="fml32_TRANSFER_p1">
  <xsd:element name="CUST_NAME" type="xsd:string" />
  <xsd:element name="CUST_ADDRESS" type="xsd:base64Binary" />
  <xsd:element name="CUST_PHONE" type="xsd:long" />
</xsd:complexType>

<xsd:complexType name="fml32_TRANSFER_p2">
  <xsd:element name="ACCOUNT_ID" type="xsd:long" />
  <xsd:element name="ACCOUNT_PW" type="xsd:base64Binary" />
</xsd:complexType>

<xsd:element name="inbuf" type="tuctype: fml32_TRANSFER_In" />

```

FML/FML32 Considerations

The following considerations apply to converting Tuxedo FML/FML32 buffers to and from XML.

- You must create an environment for converting XML to and from FML/FML32. This includes an FML field table file directory and system FML field definition files. These definitions are automatically loaded by the GWWS. FML typed buffers can be handled only if the environment is set up correctly.
- FML32 Field type `FLD_PTR` is not supported.
- The GWWS server provides strong consistency checking between the Tuxedo Service Metadata Repository FML/FML32 parameter definition and FML/FML32 definition file during start up.

If an FML/32 field is found that is not in accordance with the environment setting, or the field table field data type definition is different from the parameter data type definition in the Tuxedo Service Metadata Repository, the GWWS cannot start. Inconsistency messages are printed in the ULOG file.

- The `tmwsdlgen` command checks for consistency between the Tuxedo Service Metadata Repository FML/FML32 parameter definition and FML/FML32 definition file. If inconsistencies are found, it issue a warning and allow inconsistencies.

If an FML/32 field is found that is not in accordance with the environment setting, or the field table field data type definition is different from the parameter data type definition in the Tuxedo Service Metadata Repository, `tmwsdlgen` attempts to use Tuxedo Service Metadata Repository definitions to compose the WSDL document.

- Although the Tuxedo Service Metadata Repository may define a size attribute for “string/mbstring” typed parameters, which represents the maximum byte length that is allowed in the Tuxedo typed buffer, Oracle SALT does not expose such restriction in the generated WSDL document.
- Tuxedo primary data type “long” is indefinite between 32-bit and 64-bit scope according to different platforms. But the corresponding `xsd:long` schema type is used to describe 64-bit numeric value. The following scenario generates a SOAP fault:

The GWWS runs in 32-bit mode, and a Web service client sends a `xsd:long` typed data which exceeds the 32-bit value range.

Tuxedo X_C_TYPE Typed Buffers

Tuxedo X_C_TYPE typed buffers are equivalent, and have a similar WSDL format to, Tuxedo VIEW typed buffers. They are transparent for SOAP clients. However, even though usage is similar to the Tuxedo VIEW buffer type, SALT administrators must configure the Tuxedo Service Metadata Repository for any particular Tuxedo service that uses this buffer type.

Note: All View related considerations also take effect for X_C_TYPE typed buffer.

Tuxedo X_COMMON Typed Buffers

Tuxedo X_COMMON typed buffers are equivalent to Tuxedo VIEW typed buffers. However, they are used for compatibility between COBOL and C programs. Field types should be limited to short, long, and string.

Tuxedo X_OCTET Typed Buffers

Tuxedo X_OCTET typed buffers are equivalent to CARRAY.

Note: Tuxedo X_OCTET typed buffers can only map to `xsd:base64Binary` type. SALT 1.1 does not support MIME attachment binding for Tuxedo X_OCTET typed buffers.

Custom Typed Buffers

Oracle SALT provides a plug-in mechanism that supports custom typed buffers. You can validate the SOAP message against your own XML Schema definition, allocate custom typed buffers, and parse data into the buffers and other operations.

XML Schema built-in type `xsd:anyType` is the corresponding type for XML documents stored in a SOAP message. While using custom typed buffers, you should define and represent the actual data into an XML format and transfer between the Web service client and Tuxedo Web service stack. As with XML typed buffers, only a single root XML buffer can be stored in the SOAP body. The GWWS checks this for consistency.

For more plug-in information, see [Using Oracle SALT Plug-Ins on page 5?1](#).

XML-to-Tuxedo Data Type Mapping for External Web Services

Oracle SALT maps each `wSDL:message` as a Tuxedo FML32 buffer structure. Oracle SALT defines a set of rules for representing the XML Schema definition using FML32. To invoke external Web Services, customers need to understand the exact FML32 structure that converted from the external Web Service XML Schema definition of the corresponding message.

The following sections describe detailed WSDL message to Tuxedo FML32 buffer mapping rules:

- [XML Schema Built-In Simple Data Type Mapping](#)

- [XML Schema User Defined Data Type Mapping](#)
- [WSDL Message Mapping](#)

XML Schema Built-In Simple Data Type Mapping

Table 2-3 shows the supported XML Schema Built-In Simple Data Type and the corresponding Tuxedo FML32 Field Data Type.

Table 2-3 Supported XML Schema Built-In Simple Data Type

XML Schema Built-In Simple Type	Tuxedo FML32 Field Data Type	C/C++ Primitive Type In Tuxedo Program	Note
xsd:byte	FLD_CHAR	char	
xsd:unsignedByte	FLD_CHAR	unsigned char	
xsd:boolean	FLD_CHAR	char	Value Pattern ['T' 'F']
xsd:short	FLD_SHORT	short	
xsd:unsignedShort	FLD_SHORT	unsigned short	
xsd:int	FLD_LONG	long	
xsd:unsignedInt	FLD_LONG	unsigned long	
xsd:long	FLD_LONG	long	In a 32-bit Tuxedo program, the C primitive type long <i>cannot</i> represent all xsd:long valid value.
xsd:unsignedLong	FLD_LONG	unsigned long	In a 32-bit Tuxedo program, the C primitive type unsigned long <i>cannot</i> represent all xsd:long valid value.
xsd:float	FLD_FLOAT	float	
xsd:double	FLD_DOUBLE	double	

Table 2-3 Supported XML Schema Built-In Simple Data Type

XML Schema Built-In Simple Type	Tuxedo FML32 Field Data Type	C/C++ Primitive Type In Tuxedo Program	Note
xsd:string (and all <code>xsd:string</code> derived built-in type, such as <code>xsd:token</code> , <code>xsd:Name</code> , etc.)	FLD_STRING FLD_MBSTRING	char [] (Null-terminated string)	<code>xsd:string</code> can be optionally mapped as FLD_STRING or FLD_MBSTRING using wsdlcvt .
xsd:base64Binary	FLD_CARRAY	char []	
xsd:hexBinary	FLD_CARRAY	char []	
All other built-in data types (Data / Time related, decimal / Integer related, anyURI, QName, NOTATION)	FLD_STRING	char []	You should comply with the value pattern of the corresponding XML built-in data type. Otherwise, server-side Web service will reject the request.

The following samples demonstrate how to prepare data in a Tuxedo program for XML Schema Built-In Simple Types.

- [XML Schema Built-In Type Sample - xsd:boolean](#)
- [XML Schema Built-In Type Sample - xsd:unsignedInt](#)
- [XML Schema Built-In Type Sample - xsd:string](#)
- [XML Schema Built-In Type Sample - xsd:hexBinary](#)
- [XML Schema Built-In Type Sample - xsd:date](#)

Table 2-4 XML Schema Built-In Type Sample - xsd:boolean

XML Schema Definition
<pre><xsd:element name="flag" type="xsd:boolean" /></pre>
Corresponding FML32 Field Definition (FLD_CHAR)

Table 2-4 XML Schema Built-In Type Sample - xsd:boolean

#	Field_name	Field_type	Field_flag	Field_comments
	flag	char	-	

C Pseudo Code

```

char c_flag;
FBFR32 * request;
...
c_flag = 'T'; /* Set True for boolean data */
Fadd32( request, flag, (char *)&c_flag, 0);
    
```

Table 2-5 XML Schema Built-In Type Sample - xsd:unsignedInt

XML Schema Definition

```

<xsd:element name="account" type="xsd:unsignedInt" />
    
```

Corresponding FML32 Field Definition (FLD_LONG)

#	Field_name	Field_type	Field_flag	Field_comments
	account	long	-	

C Pseudo Code

```

unsigned long acc;
FBFR32 * request;
...
acc = 102377; /* Value should not exceed value scope of unsigned int*/
Fadd32( request, account, (char *)&acc, 0);
    
```

Table 2-6 XML Schema Built-In Type Sample - xsd:string

XML Schema Definition

```

<xsd:element name="message" type="xsd:string" />
    
```

Corresponding FML32 Field Definition (FLD_MBSTRING)

#	Field_name	Field_type	Field_flag	Field_comments
	message	mbstring	-	

Table 2-6 XML Schema Built-In Type Sample - xsd:string**C Pseudo Code**

```

FBFR32 * request;
FLDLLEN32 len, mbsize = 1024;
char * msg, * mbmsg;
msg = calloc( ... ); mbmsg = malloc(mbsize);
...
strncpy(msg, "...", len); /* The string is UTF-8 encoding */
Fmbpack32("utf-8", msg, len, mbmsg, &mbsize, 0); /* prepare mbstring*/
Fadd32( request, message, mbmsg, mbsize);

```

Table 2-7 XML Schema Built-In Type Sample - xsd:hexBinary**XML Schema Definition**

```
<xsd:element name="mem_snapshot" type="xsd:hexBinary" />
```

Corresponding FML32 Field Definition (FLD_MBSTRING)

#	Field_name	Field_type	Field_flag	Field_comments
	mem_snapshot	carray	-	

C Pseudo Code

```

FBFR32 * request;
FLDLLEN32 len;
char * buf;
buf = calloc( ... );
...
memcpy(buf, "...", len); /* copy the original memory */
Fadd32( request, mem_snapshot, buf, len);

```

Table 2-8 XML Schema Built-In Type Sample - xsd:date**XML Schema Definition**

```
<xsd:element name="IssueDate" type="xsd:date" />
```

Table 2-8 XML Schema Built-In Type Sample - xsd:date

Corresponding FML32 Field Definition (FLD_STRING)				
#	Field_name	Field_type	Field_flag	Field_comments
	IssueDate	string	-	

C Pseudo Code
<pre> FBFR32 * request; char date[32]; ... strcpy(date, "2007-06-04+8:00"); /* Set the date value correctly */ Fadd32(request, IssueDate, date, 0); </pre>

XML Schema User Defined Data Type Mapping

Table 2-9 lists the supported XML Schema User Defined Simple Data Type and the corresponding Tuxedo FML32 Field Data Type.

Table 2-9 Supported XML Schema User Defined Data Type

XML Schema User Defined Data Type	Tuxedo FML32 Field Data Type	C/C++ Primitive Type In Tuxedo Program	Note
<xsd:anyType>	FLD_MBSTRING	char []	Tuxedo Programmer should prepare entire XML document enclosing with the element tag.
<xsd:simpleType> derived from built-in primitive simple data types	Equivalent FML32 Field Type of the primitive simple type (see Table 2-3)	Equivalent C Primitive Data Type of the primitive simple type (see Table 2-3)	Facets defined with <xsd:restriction> are not enforced at Tuxedo side.
<xsd:simpleType> defined with <xsd:list>	FLD_MBSTRING	char []	Same as <xsd:anyType>. The Schema compliancy is not enforced at Tuxedo side.

Table 2-9 Supported XML Schema User Defined Data Type

XML Schema User Defined Data Type	Tuxedo FML32 Field Data Type	C/C++ Primitive Type In Tuxedo Program	Note
<xsd:simpleType> defined with <xsd:union>	FLD_MBSTRING	char []	Same as <xsd:anyType>. The Schema compliancy is not enforced at Tuxedo side.
<xsd:complexType> defined with <xsd:simpleContent>	FLD_MBSTRING	char []	Same as <xsd:anyType>. The Schema compliancy is not enforced at Tuxedo side.
<xsd:complexType> defined with <xsd:complexContent>	FLD_MBSTRING	char []	Same as <xsd:anyType>. The Schema compliancy is not enforced at Tuxedo side.
<xsd:complexType> defined with shorthand <xsd:complexContent>, sub-elements composited with sequence or all	FLD_FML32	FBFR32 * embedded fml32 buffer	Each sub-element of the complex type is defined as an embedded FML32 field.
<xsd:complexType> defined with shorthand <xsd:complexContent>, sub-elements composited with choice	FML_FML32	FBFR32 * embedded fml32 buffer	Each sub-element of the complex type is defined as an embedded FML32 field. Tuxedo programmer should only add one sub field into the fml32 buffer.

The following samples demonstrate how to prepare data in a Tuxedo program for XML Schema User Defined Data Types:

- [XML Schema User Defined Type Sample - xsd:simpleType Derived from Primitive Simple Type](#)

- [XML Schema User Defined Type Sample - xsd:simpleType Defined with xsd:list](#)

Table 2-10 XML Schema User Defined Type Sample - xsd:simpleType Derived from Primitive Simple Type

XML Schema Definition				
<pre><xsd:element name="Grade" type="Alphabet" /> <xsd:simpleType name="Alphabet"> <xsd:restriction base="xsd:string"> <xsd:maxLength value="1" /> <xsd:pattern value="[A-Z]" /> </xsd:restriction> </xsd:simpleType></pre>				
Corresponding FML32 Field Definition (FLD_STRING)				
#	Field_name	Field_type	Field_flag	Field_comments
	Grade	string	-	
C Pseudo Code				
<pre>char grade[2]; FBFR32 * request; ... grade[0] = 'A'; grade[1] = '\0'; Fadd32(request, Grade, (char *)grade, 0);</pre>				

Table 2-11 XML Schema User Defined Type Sample - xsd:simpleType Defined with xsd:list

XML Schema Definition (Target Namespace "urn:sample.org")				
<pre><xsd:element name="Users" type="namelist" /> <xsd:simpleType name="namelist"> <xsd:list itemType="xsd:NMTOKEN"> </xsd:simpleType></pre>				
Corresponding FML32 Field Definition (FLD_MBSTRING)				
#	Field_name	Field_type	Field_flag	Field_comments
	Users	mbstring	-	

Table 2-11 XML Schema User Defined Type Sample - xsd:simpleType Defined with xsd:list**C Pseudo Code**

```

char * user[5];
char users[...];
char * mbpacked;
FLDLEN32 mbsize = 1024;
FBFR32 * request;
...
sprintf(users, "<n1:Users xmlns:n1=\"urn:sample.org\">");
for ( i = 0 ; i < 5 ; i++ ) {
    strcat(users, user[i]);
    strcat(users, " ");
}
strcat(users, "</n1:Users>");
...
mbpacked = malloc(mbsize);
/* prepare mbstring*/
Fmbpack32("utf-8", users, strlen(users), mbpacked, &mbsize, 0);
Fadd32( request, Users, mbpacked, mbsize);

```

WSDL Message Mapping

Tuxedo FML32 buffer type is always used in mapping WSDL messages.

[Table 2-12](#) lists the WSDL message mapping rules defined by Oracle SALT.

Table 2-12 WSDL Message Mapping Rules

WSDL Message Definition	Tuxedo Buffer/Field Definition	Note
<wsdl:input> message	Tuxedo Request Buffer (Input buffer)	
<wsdl:output> message	Tuxedo Response Buffer with TPSUCCESS (Output buffer)	
<wsdl:fault> message	Tuxedo Response Buffer with TPFAIL (error buffer)	

Table 2-12 WSDL Message Mapping Rules

WSDL Message Definition	Tuxedo Buffer/Field Definition	Note
Each message part defined in <code><wsdl:input></code> or <code><wsdl:output></code>	Mapped as top level field in the Tuxedo FML32 buffer. Field type is the equivalent FML32 field type of the message part XML data type. (See Table 2-3 and Table 2-9)	
<code><faultcode></code> in SOAP 1.1 fault message	Mapped as a fixed top level FLD_STRING field (<code>faultcode</code>) in the Tuxedo error buffer: <code>faultcode string - -</code>	This mapping rule applies for SOAP 1.1 only.
<code><faultstring></code> in SOAP 1.1 fault message	Mapped as a fixed top level FLD_STRING field (<code>faultstring</code>) in the Tuxedo error buffer: <code>faultstring string - -</code>	This mapping rule applies for SOAP 1.1 only.
<code><faultactor></code> in SOAP 1.1 fault message	Mapped as a fixed top level FLD_STRING field (<code>faultactor</code>) in the Tuxedo error buffer: <code>faultactor string - -</code>	This mapping rule applies for SOAP 1.1 only.
<code><Code></code> in SOAP 1.2 fault message	Mapped as a fixed top level FLD_FML32 field (<code>Code</code>) in the Tuxedo error buffer, which containing two fixed sub FLD_STRING fields (<code>Value</code> and <code>Subcode</code>): <code>Code fml32 - -</code> <code>Value string - -</code> <code>Subcode string - -</code>	This mapping rule applies for SOAP 1.2 only.
<code><Reason></code> in SOAP 1.2 fault message	Mapped as a fixed top level FLD_FML32 field (<code>Reason</code>) in the Tuxedo error buffer, which containing zero or more fixed sub FLD_STRING field (<code>Text</code>): <code>Reason fml32 - -</code> <code>Text string - -</code>	This mapping rule applies for SOAP 1.2 only.
<code><Node></code> in SOAP 1.2 fault message	Mapped as a fixed top level FLD_STRING field (<code>Node</code>) in the Tuxedo error buffer: <code>Node string - -</code>	This mapping rule applies for SOAP 1.2 only.

Table 2-12 WSDL Message Mapping Rules

WSDL Message Definition	Tuxedo Buffer/Field Definition	Note
<Role> in SOAP 1.2 fault message	Mapped as a fixed top level FLD_STRING field (Role) in the Tuxedo error buffer: Role string - -	This mapping rule applies for SOAP 1.2 only.
<detail> in SOAP fault message	Mapped as a fixed top level FLD_FML32 field in the Tuxedo error buffer: detail fml32 - -	This mapping rule applies for both SOAP 1.1 and SOAP 1.2.
Each message part defined in <wsdl: fault>	Mapped as a sub field of “detail” field in the Tuxedo FML32 buffer. Field type is the equivalent FML32 field type of the message part XML data type. (See Table 2-3 and Table 2-9)	This mapping rule applies for both SOAP 1.1 and SOAP 1.2.

Data Type Mapping and Message Conversion

Web Service Client Programming

This section contains the following topics:

- [Overview](#)
- [Oracle SALT Web Service Client Programming Tips](#)
- [Web Service Client Programming References](#)

Overview

Oracle SALT is a configuration-driven product that publishes existing Tuxedo application services as industry-standard Web services. From a Web services client-side programming perspective, Oracle SALT used in conjunction with the Oracle Tuxedo framework is a standard Web service provider. You only need to use the Oracle SALT WSDL file to develop a Web service client program.

To develop a Web service client program, do the following steps:

1. Generate or download the Oracle SALT WSDL file. For more information, see [Configuring Oracle SALT](#) in the *Oracle SALT Administration Guide*.
2. Use a Web service client-side toolkit to parse the SALT WSDL document and generate client stub code. For more information, see [Oracle SALT Web Service Client Programming Tips](#).
3. Write client-side application code to invoke a Oracle SALT Web service using the functions defined in the client-generated stub code.
4. Compile and run your client application.

Oracle SALT Web Service Client Programming Tips

This section provides some useful client-side programming tips for developing Web service client programs using the following Oracle SALT-tested programming toolkits:

- [Oracle WebLogic Web Service Client Programming Toolkit](#)
- [Apache Axis for Java Web Service Client Programming Toolkit](#)
- [Microsoft .NET Web Service Client Programming Toolkit](#)

For more information, see *Oracle SALT Interoperability*.

Notes: You can use any SOAP toolkit to develop client software.

The sample directories for the listed toolkits can be found *after* Oracle SALT is installed.

Oracle WebLogic Web Service Client Programming Toolkit

WebLogic Server provides the `clientgen` utility which is a built-in application server component used to develop Web service client-side java programs. The invocation can be issued from standalone java program and server instances. For more information, see http://edocs.bea.com/wls/docs91/webserv/client.html#standalone_invoke.

Besides traditional synchronous message exchange mode, Oracle SALT also supports asynchronous and reliable Web service invocation using WebLogic Server. Asynchronous communication is defined by the WS-Addressing specification. Reliable message exchange conforms to the WS-ReliableMessaging specification.

Tip: Use the WebLogic specific WSDL document for HTTP MIME attachment support.

Oracle SALT can map Tuxedo CARRAY data to SOAP request MIME attachments. This is beneficial when the binary data stream is large since MIME binding does not need additional encoding wrapping. This can help save CPU cycles and network bandwidth.

Another consideration, in an enterprise service oriented environment, is that binary data might be used to guide high-level data routing and transformation work. Encoded data can be problematic. To enable the MIME data binding for Tuxedo CARRAY data, a special flag must be specified in the WSDL document generation options; both for online downloading and using the `tmwsdlgen` command utility.

Online Download:

`http://salt.host:portnumber//wsdl?mappolicy=raw&toolkit=wls`

tmwsdlgen Utility

```
tmwsdlgen -c WSDL_FILE -m raw -t wls
```

Apache Axis for Java Web Service Client Programming Toolkit

Oracle SALT supports the AXIS `wsdl2java` utility which generates java stub code from the WSDL document. The AXIS Web service programming model is similar to WebLogic.

Tip: 1. Use the AXIS specific WSDL document for HTTP MIME attachment support.

Oracle SALT supports HTTP MIME transportation for Tuxedo CARRAY data. A special option must be specified for WSDL online downloading and the `tmwsdlgen` utility.

Online Download:

```
http://salt.host:portnumber//wsdl?mappolicy=raw&toolkit=axis
```

tmwsdlgen Utility

```
tmwsdlgen -c WSDL_FILE -m raw -t axis
```

Tip: 2. Disable multiple-reference format in AXIS when RPC/encoded style is used.

AXIS may send a multi-reference format SOAP message when RPC/encoded style is specified for the WSDL document. Oracle SALT does not support multiple-reference format. You can disable AXIS multiple-reference format as shown in [Listing 3-1](#):

Listing 3-1 Disabling AXIS Multiple-Reference Format

```
TuxedoWebServiceLocator service = new TuxedoWebServiceLocator();
service.getEngine().setOption("sendMultiRefs", false);
```

Tip: 3. Use Apache Sandsha project with Oracle SALT for WS-ReliableMessaging communication.

Interoperability was tested for WS-ReliableMessaging between Oracle SALT and the Apache Sandsha project. The Sandsha asynchronous mode and `send offer` must

be set in the code.

A sample Apache Sandesha asynchronous mode and `send offer` code example is shown in [Listing 3-2](#):

Listing 3-2 Sample Apache Sandesha Asynchronous Mode and “send offer” Code example

```
/* Call the service */
    TuxedoWebService service = new TuxedoWebServiceLocator();

    Call call = (Call) service.createCall();
    SandeshaContext ctx = new SandeshaContext();

    ctx.setAcksToURL("http://127.0.0.1:" + defaultClientPort +
"/axis/services/RMService");
    ctx.setReplyToURL("http://127.0.0.1:" + defaultClientPort +
"/axis/services/RMService");
    ctx.setSendOffer(true);
    ctx.initCall(call, targetURL, "urn:wsm:simpapp",
Constants.ClientProperties.IN_OUT);

    call.setUseSOAPAction(true);
    call.setSOAPActionURI("ToUpperWS");
    call.setOperationName(new
javax.xml.namespace.QName("urn:pack:simpappsimpapp_typedef:salt11",
"ToUpperWS"));
    call.addParameter("inbuf", XMLType.XSD_STRING, ParameterMode.IN);
    call.setReturnType(org.apache.axis.encoding.XMLType.XSD_STRING);

    String input = new String();
    String output = new String();
    int i;
    for (i = 0; i < 3; i++) {
        input = "request" + "_" + String.valueOf(i);

        System.out.println("Request: "+input);
```

```

        output = (String) call.invoke(new Object[]{input});
        System.out.println("Reply:" + output);
    }

ctx.setLastMessage(call);
    input = "request" + "_" + String.valueOf(i);
    System.out.println("Request:"+input);
    output = (String) call.invoke(new Object[]{input});

```

Microsoft .NET Web Service Client Programming Toolkit

Microsoft .Net 1.1/2.0 provides `wsdl.exe` in the .Net SDK package. It is a free development Microsoft toolkit. In the Oracle SALT `simpapp` sample, a .Net program is provided in the `simpapp/dnetclient` directory.

.Net Web service programming is easy and straightforward. Use the `wsdl.exe` utility and the Oracle SALT WSDL document to generate the stub code, and then reference the .Net object contained in the stub code/binary in business logic implementations.

Tip: 1. Do not use .Net program MIME attachment binding for CARRAY.

Microsoft does not support SOAP communication MIME binding. Avoid using the WSDL document with MIME binding for CARRAY in .Net development.

Oracle SALT supports `base64Binary` encoding for CARRAY data (the default WSDL document generation.)

Tip: 2. Some RPC/encoded style SOAP messages are not understood by the GWWS server.

When the Oracle SALT WSDL document is generated using RPC/encoded style, .Net sends out SOAP messages containing `soapenc:arrayType`. Oracle SALT does not support `soapenc:arrayType` using RPC/encoded style. A sample RPC/encoded style-generated WSDL document is shown in [Listing 3-3](#).

Listing 3-3 Sample RPC/encoded Style-Generated WSDL document

```

<wsdl:types>
    <xsd:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
targetNamespace="urn:pack.TuxAll_typedef.salt11">
        <xsd:complexType name="fml_TFML_In">
            <xsd:sequence>
                <xsd:element maxOccurs="60"
minOccurs="60" name="tflong" type="xsd:long"></xsd:element>
                <xsd:element maxOccurs="80"
minOccurs="80" name="tffloat" type="xsd:float"></xsd:element>
            </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="fml_TFML_Out">
            ...
        </xsd:complexType>
    </xsd:schema>
</wsdl:types>

```

Workaround: Use Document/literal encoded style for .Net client as recommended by Microsoft.

Tip: 3. Error message regarding `xsd:base64Binary` in RPC/encoded style.

If `xsd:base64Binary` is used in the Oracle SALT WSDL document in RPC/encoded style, `wsdl.exe` can generate stub code, but the client program might report a runtime error as follows:

```
System.InvalidOperationException: 'base64Binary' is an invalid value for the SoapElementAttribute.DataType property. The property may only be specified for primitive types.
```

Workaround: This is a .Net framework issue.

Use Document/literal encoded style for .Net client as recommended by Microsoft.

Web Service Client Programming References

Online References

- Oracle WebLogic 10.0 Web Service Client Programming References
[Invoking a Web service from a Stand-alone Client: Main Steps](#)
- Apache Axis 1.3 Web Service Client Programming References
[Consuming Web Services with Axis](#)
[Using WSDL with Axis](#)
- Microsoft .NET Web Service Programming References
[Building Web Services](#)

Oracle Tuxedo ATMI Programming for Web Services

This chapter contains the following topics:

- [Overview](#)
- [Converting WSDL Model Into Tuxedo Model](#)
- [Invoking SALT Proxy Services](#)

Overview

Oracle SALT allows you to import external Web Services into Tuxedo Domains. To import external Web services into Tuxedo application, a WSDL file must first be loaded and converted. The Oracle SALT WSDL conversion utility, `wsdlcvt`, translates each `wsdl:operation` into a Oracle SALT proxy service. The translated SALT proxy service can be invoked directly through standard Tuxedo ATMI functions.

Oracle SALT proxy service calls are sent to the GWWS server. The request is translated from Tuxedo typed buffers into the SOAP message, and then sent to the corresponding external Web Service. The response from an external Web Service is translated into Tuxedo typed buffers and returned to the Tuxedo application. The GWWS acts as the proxy intermediary.

If an error occurs during the service call, the GWWS server sets the error status using `tperrno`, which can be retrieved by Tuxedo applications. This enables you to detect and handle the SALT proxy service call error status.

Converting WSDL Model Into Tuxedo Model

Oracle SALT provides a WSDL conversion utility, [wsdlcvt](#), that converts external WSDL files into Tuxedo specific definition files so that you can develop Tuxedo ATMI programs to access services defined in the WSDL file.

WSDL-to-Tuxedo Object Mapping

Oracle SALT converts WSDL object models into Tuxedo models using the following rules:

- Only SOAP over HTTP binding are supported, each binding is defined and saved as a `WSBinding` object in the WSDL file.
- Each operation in the SOAP bindings is mapped as one Tuxedo style service, which is also called a SALT proxy service. The operation name is used as the Tuxedo service name and indexed in the Tuxedo Service Metadata Repository.

Note: If the operation name exceeds the Tuxedo service name length limitation (255 characters), you must manually set a unique short Tuxedo service name in the metadata repository and set the `<Service>` `tuxedoRef` attribute in the WSDL file.

For more information, see [Oracle SALT Web Service Definition File Reference](#) in the *Oracle SALT Reference Guide*.

- Other Web service external application protocol information is saved in the generated WSDL file (including SOAP protocol version, SOAP message encoding style, accessing endpoints, and so).
- XML Schema definitions embedded in the WSDL file are copied and saved in separate `.xsd` files.
- Each `wsdl:operation` object and its input/output message details are converted as a Tuxedo service definition conforms to Tuxedo Service Metadata Repository input syntax.

[Table 4-1](#) lists detailed mapping relationships between the WSDL file and Tuxedo definition files.

Table 4-1 WSDL Model / Tuxedo Model Mapping Rules

WSDL Object	Tuxedo/SALT Definition File	Tuxedo/SALT Definition Object
/wsdl:binding	SALT Web Service Definition File (WSDF)	/WSBinding
/wsdl:portType		/WSBinding/Servicegroup
/wsdl:binding/soap:binding		/WSBinding/SOAP
/wsdl:portType/operation	Metadata Input File (MIF)	/WSBinding/service
/wsdl:types/xsd:schema	FML32 Field Definition Table	Field name type

Invoking SALT Proxy Services

The following sections include information on how to invoke the converted SALT proxy service from a Tuxedo application:

- [Oracle SALT Supported Communication Pattern](#)
- [Tuxedo Outbound Call Programming: Main Steps](#)
- [Managing Error Code Returned from GWWS](#)
- [Handling Fault Messages in a Tuxedo Outbound Application](#)

Oracle SALT Supported Communication Pattern

Oracle SALT only supports the Tuxedo Request/Response communication patterns for outbound service calls. A Tuxedo application can request the SALT proxy service using the following communication Tuxedo ATMI:

- `tpcall(1)` / `tpacall(1)` / `tpgetreply(1)`

These basic ATMI functions can be called with a Tuxedo typed buffer as input parameter. The return of the call will also carry a Tuxedo typed buffer. All these buffers will conform to the converted outside Web service interface. `tpacall/tpgetreply` is not related to SOAP async communication.

- `tpforward(1)`

Tuxedo server application can use this function to forward a Tuxedo request to a specified SALT proxy service. The response buffer is sent directly to client application's response queue as if it's a traditional native Tuxedo service.

- `TMQFORWARD` enabled queue-based communication.

Tuxedo system server `TMQFORWARD` can accept queued requests and send them to Oracle SALT proxy services that have the same name as the queue.

Oracle SALT does not support the following Tuxedo communication patterns:

- Conversational communication
- Event-based communication

Tuxedo Outbound Call Programming: Main Steps

When the GWWS is booted and Oracle SALT proxy services are advertised, you can create a Tuxedo application to call them. To develop a program to access SALT proxy services, do the following:

- Check the Tuxedo Service Metadata Repository definition to see what the SALT proxy service interface is.
- Locate the generated FML32 field table files. Modify the FML32 field table to eliminate conflicting field names and assign a valid base number for the index.

Note: The `wsdlcvt` generated FML32 field table files are always used by GWWS. you must make sure the field name is unique at the system level. If two or more fields are associated with the same field name, change the field name. Do not forget to change Tuxedo Service Metadata Repository definition accordingly.

The base number of field index in the generated FML32 field table must be changed from the invalid default value to a correct number to ensure all field index in the table is unique at the entire system level.

- Generate FML32 header files with `mkfldhdr32(1)`.
- Boot the GWWS with correct FML32 environment variable settings.
- Write a skeleton C source file for the client to call the outbound service (refer to Tuxedo documentation and the Tuxedo Service Metadata Repository generated pseudo-code if necessary). You can use `tpcall(1)` or `tpacall(1)` for synchronous or asynchronous communication, depending on the requirement.

- For FML32 buffers, you need to add each FML32 field (conforming to the corresponding Oracle SALT proxy service input buffer details) defined in the Tuxedo Service Metadata Repository, including FML32 field sequence and occurrence. The client source may include the generated header file to facilitate referencing the field name.
- Get input buffer ready, user can handle the returned buffer, which should be of the type defined in Metadata.
- Compile the source to generate executable.
- Test the executable.

Managing Error Code Returned from GWWS

If the GWWS server encounters an error accessing external Web services, `tperrno` is set accordingly so the Tuxedo application can diagnose the failure. [Table 4-2](#) lists possible Oracle SALT proxy service `tperrno` values.

Table 4-2 Error Code Returned From GWWS/Tuxedo Framework

TPERRNO	Possible Failure Reason
TPENOENT	Requested SALT proxy service is not advertised by GWWS
TPESVCERR	The HTTP response message returned from external Web service application is not valid The SOAP response message returned from external Web service application is not well-formed.
TPEPERM	Authentication failure.
TPEITYPE	Message conversion failure when converting Tuxedo request typed buffer into XML payload of the SOAP request message.
TPEOTYPE	Message conversion failure when converting XML payload of the SOAP response message into Tuxedo response typed buffer.
TPEOS	Request is rejected because of system resource limitation
TPETIME	Timeout occurred. This timeout can either be a BBL blocktime, or a SALT outbound call timeout.

Table 4-2 Error Code Returned From GWWS/Tuxedo Framework

TPERRNO	Possible Failure Reason
TPSVCFAIL	External Web service returns SOAP fault message
TPESYSTEM	GWWS internal errors. Check ULOG for more information.

Handling Fault Messages in a Tuxedo Outbound Application

All rules listed in used to map WSDL input/output message into Tuxedo Metadata inbuf/outbuf definition. WSDL file default message can also be mapped into Tuxedo Metadata errbuf, with some amendments to the rules:

Rules for fault mapping:

There are two modes for mapping Metadata `errbuf` into SOAP Fault messages: Tux Mode and XSD Mode.

- Tux Mode is used to convert Tuxedo original error buffers returned with `TPFAIL`. The error buffers are converted into XML payload in the SOAP fault `<detail>` element.
- XSD Mode is used to represent SOAP fault and WSDL file fault messages defined with Tuxedo buffers. The mapping rule includes:
 - Each service in XSD mode (`servicemode=webservice`) always has an `errbuf` in Metadata, with `type=FML32`.
 - `errbuf` is a FML32 buffer. It is a complete description of the SOAP:Fault message that may appear in correspondence (which is different for SOAP 1.1 and 1.2). The `errbuf` definition content is determined by the SOAP version and WSDL fault message both.
 - Parameter `detail/Detail (1.1/1.2)` is an FML32 field that represents the `wsdl:part` defined in a `wsdl:fault` message (when `wsdl:fault` is present). Each part is defined as a `param(field)` in the FML32 field. The mapping rules are the same as for input/output buffer. The difference is that each `param requiredcount` is 0, which means it may not appear in the SOAP fault message.
 - Other elements that appear in `soap:fault` message are always defined as a field in `errbuf`, with `requiredcount` equal to 1 or 0 (depending on whether the element is required or optional).
 - Each part definition in the Metadata controls converting a `<detail>` element in the soap fault message into a field in the error buffer.

Table 4-3 lists the outbound SOAP fault errbuf definitions.

Table 4-3 Outbound SOAP Fault Errbuf Definition

Meta Parameter	SOAP Version	Type	Required	Memo
faultcode	1.1	string	Yes	
faultstring	1.1	string	Yes	
faultactor	1.1	string	No	
detail	1.1	fml32	No	If no wsdl: fault is defined, this field will contain an XML field.
Code	1.2	fml32	Yes	Contain Value and optional Subcode
Reason	1.2	fml32	Yes	Contains multiple Text
Node	1.2	string	No	
Role	1.2	string	No	
Detail	1.2	fml32	No	same as detail field

Using Oracle SALT Plug-Ins

This chapter contains the following topics:

- [Understanding Oracle SALT Plug-Ins](#)
- [Programming Message Conversion Plug-ins](#)
- [Programming Outbound Authentication Plug-Ins](#)

Understanding Oracle SALT Plug-Ins

The Oracle SALT [GWWS](#) server is a configuration-driven process which, for most basic Web service applications, does not require any programming tasks. However, Oracle SALT functionality can be enhanced by developing plug-in interfaces which utilize custom typed buffer data and customized shared libraries to extend the GWWS server.

A plug-in interface is a set of functions exported by a shared library that can be loaded and invoked by GWWS processes to achieve special functionality. Oracle SALT provides a plug-in framework as a common interface for defining and implementing a plug-in interface. Plug-in implementation is carried out by a shared library which contains the actual functions. The plug-in implementation library is configured in the [SALT Deployment file](#) and is loaded dynamically during GWWS server startup.

Plug-In Elements

Four plug-in elements are required to define a plug-in interface:

- [Plug-In ID](#)

- [Plug-In Name](#)
- [Plug-In Implementation Functions](#)
- [Plug-In Register Functions](#)

Plug-In ID

The plug-in ID element is a string used to identify a particular plug-in interface function. Multiple plug-in interfaces can be grouped with the same Plug-in ID for a similar function. Plug-in ID values are predefined by Oracle SALT. Arbitrary string values are not permitted.

Oracle SALT 10gR3 supports the `P_CUSTOM_TYPE` and `P_CREDENMAP` plug-in ID, which is used to define plug-in interfaces for custom typed buffer data handling, and map Tuxedo user ID and group ID into username/password that HTTP Basic Authentication needs.

Plug-In Name

The plug-in Name differentiates one plug-in implementation from another within the same Plug-in ID category.

For the `P_CUSTOM_TYPE` Plug-in ID, the plug-in name is used to indicate the actual custom buffer type name. When the GWWS server attempts to convert data between Tuxedo custom typed buffers and an XML document, the plug-in name is the key element that searches for the proper plug-in interface.

Plug-In Implementation Functions

Actual business logic should reflect the necessary functions defined in a plug-in vtable structure. Necessary functions may be different for different plug-in ID categories.

For the `P_CREDENMAP` ID category, one function needs to be implemented:

- `int (* gwws_pi_map_http_basic) (char * domain, char * realm, char * t_userid, char * t_grpid, Cred_UserPass * credential);`

For more information, see [Programming Outbound Authentication Plug-Ins](#).

Plug-In Register Functions

Plug-in Register functions are a set of common functions (or rules) that a plug-in interface must implement so that the GWWS server can invoke the plug-in implementation. Each plug-in interface must implement three register function. These functions are:

- [Information Providing Function](#)
- [Initiating Function](#)
- [Exiting Function](#)
- [vtable Setting Function](#)

Information Providing Function

This function is optional. If it is used, it is first invoked after the plug-in shared library is loaded during GWWS server startup. If you want to implement more than one interface in one plug-in library, you must implement this function and return the counts, IDs, and names of the interfaces in the library.

Returning a 0 value indicates the function has executed successfully. Returning a value other than 0 indicates failure. If this function fails, the plug-in is not loaded and the GWWS server will not start.

The function uses the following syntax:

```
int _ws_pi_get_Id_and_Names(int * count, char **ids, char **names);
```

You must return the total count of implementation in the library in argument `count`. The arguments `IDs` and `names` should contain all implemented interface `IDs` and `names`, separated by a semicolon “;”.

Initiating Function

The initiating function is invoked after all the implemented interfaces in the plug-in shared library are determined. You can initialize data structures and set up global environments that can be used by the plug-ins.

Returning a 0 value indicates the initiating function has executed successfully. Returning a value other than 0 indicates initiation has failed. If plug-in interface initiation fails, the GWWS server will not start.

The initiating function uses the following syntax:

```
int _ws_pi_init_@ID@_@Name@(char * params, void **priv_ptr);
```

`@ID@` indicates the actual plug-in ID value. `@Name@` indicates the actual plug-in name value. For example, the initiating function of a plug-in with `P_CUSTOM_TYPE` as a plug-in ID and `MyType` as a plug-in name is: `_ws_pi_init_P_CUSTOM_TYPE_MyType (char * params, void **priv_ptr)`.

Exiting Function

The exiting function is called before closing the plug-in shared library when the GWWS server shuts down. You should release all reserved plug-in resources.

The exiting function uses the following syntax:

```
int _ws_pi_exit_@ID@_@Name@(void * priv);
```

@ID@ indicates the actual plug-in ID value. @Name@ indicates the actual plug-in name value. For example, the initiating exiting function name of a plug-in with `P_CUSTOM_TYPE` as a plug-in ID and `MyType` as a plug-in name is: `_ws_pi_exit_P_CUSTOM_TYPE_MyType(void * priv)`.

vtable Setting Function

`vtable` is a particular C structure that stores the necessary function pointers for the actual business logic of a plug-in interface. In other words, a valid plug-in interface must implement all the functions defined by the corresponding `vtable`.

The `vtable` setting function uses the following syntax:

```
int _ws_pi_set_vtbl_@ID@_@Name@(void * priv);
```

@ID@ indicates the actual plug-in ID value. @Name@ indicates the actual plug-in name value. For example, the `vtable` setting function of a plug-in with `P_CUSTOM_TYPE` as a plug-in ID and `MyType` as a plug-in name is: `_ws_pi_set_vtbl_P_CUSTOM_TYPE_MyType(void * priv)`.

The `vtable` structures may be different for different plug-in ID categories. For the Oracle SALT 10gR3 release, `P_CUSTOM_TYPE` and `P_CREDENMAP` are the only valid plug-in IDs.

The `vtable` structures for available plug-in interfaces are shown in [Listing 5-1](#).

Listing 5-1 VTable Structure

```
struct credmap_vtable {
    int (* gwws_pi_map_http_basic) (char * domain, char * realm, char *
t_userid, char * t_grpid, Cred_UserPass * credential); /* used for HTTP
Basic Authentication */
    /* for future use */
    void * unused_1;
    void * unused_2;
    void * unused_3;
};
```

`struct credmap_vtable` indicates that one function need to be implemented for a `P_CREDENMAP` plug-in interface. For more information, see [Programming Outbound Authentication Plug-Ins](#).

The function input parameter `void * priv` points to a concrete vtable instance. You should set the vtable structure with the actual functions within the vtable setting function.

An example of setting the vtable structure with the actual functions within the vtable setting function is shown in [Listing 5-2](#).

Listing 5-2 Setting the vtable Structure with Actual functions within the vtable Setting Function

```
int _DLLEXPORT_ _ws_pi_set_vtbl_P_CREDENMAP_TEST (void * vtbl)
{
    struct credmap_vtable * vtable;
    if ( ! vtbl )
        return -1;

    vtable = (struct credmap_vtable *) vtbl;

    vtable->gws_pi_map_http_basic = Credmap_HTTP_Basic;
    return 0;
}
```

Developing a Plug-In Interface

To develop a comprehensive plug-in interface, do the following steps:

1. Develop a shared library to implement the plug-in interface
2. Define the plug-in interface in the SALT configuration file

Developing a Plug-In Shared Library

To develop a plug-in shared library, do the following steps:

1. Write C language plug-in implementation functions for the actual business logic. These functions are not required to be exposed from the shared library. For more information, see [Plug-In Implementation Functions](#).
2. Write C language plug-in register functions that include: the initiating function, the exiting function, the vtable setting function, and the information providing function if necessary. These register functions need to be exported so that they can be invoked from the GWWS server. For more information, see [Plug-In Register Functions](#).
3. Compile all the above functions into one shared library.

Defining a Plug-In interface in SALT configuration file

To define a plug-in shared library that is loaded by the GWWS server, the corresponding plug-in library path must be configured in the SALT deployment file. For more information, see [Setting Up a Oracle SALT Application](#) in the *Oracle SALT Administration Guide*.

An example of how to define plug-in information in the Oracle SALT deployment file is shown in [Listing 5-3](#).

Listing 5-3 Defined Plug-In in the Oracle SALT Deployment File

```
<?xml version="1.0" encoding="UTF-8"?>
<Deployment xmlns="http://www.bea.com/Tuxedo/SALTDEPLOY/2007">
    . . . . .
    . . . . .
    <System>
        <Plugin>
            <Interface
                id="P_CREDENMAP"
                name="TEST"
                library="credmap_plugin.dll" />
        </Plugin>
    </System>
</Deployment>
```

Notes: To define multiple plug-in interfaces, multiple <Interface> elements must be specified. Each <Interface> element indicates one plug-in interface.

Multiple plug-in interfaces can be built into one shared library file.

Programming Message Conversion Plug-ins

Oracle SALT defines a complete set of default data type conversion rules to convert between Tuxedo buffers and SOAP message payloads. However, the default data type conversion rules may not meet all your needs in transforming SOAP messages into Tuxedo typed buffers or vice versa. To accommodate special application requirements, Oracle SALT supports customized message level conversion plug-in development to extend the default message conversion.

Note: The SALT 10gR3 Message Conversion Plug-in is an enhanced successor of the SALT 1.1 Custom Buffer Type Conversion Plug-in.

The following topics are included in this section:

- [How Message Conversion Plug-ins Work on page 5?7](#)
- [When Do We Need Message Conversion Plug-in on page 5?10](#)
- [Developing a Message Conversion Plug-in Instance on page 5?12](#)
- [SALT 1.1 Custom Buffer Type Conversion Plug-in Compatibility on page 5?16](#)

How Message Conversion Plug-ins Work

Message Conversion Plug-in is a SALT supported Plug-in defined within the SALT plug-in framework. All Message Conversion Plug-in instances have the same [Plug-In ID](#), “P_CUSTOM_TYPE“. Each particular Message Conversion Plug-in instance may implement two functions, one is used to convert SOAP message payloads to Tuxedo buffers, and the other is used to convert Tuxedo buffers to SOAP message payloads. These two function prototypes are defined in [Listing 5-4](#).

Listing 5-4 vtable structure for SALT Plug-in “P_CUSTOM_TYPE” (C Language)

```
/* custtype_pi_ex.h */
struct custtype_vtable {
    CustomerBuffer * (* soap_in_tuxedo__CUSTBUF) (void * xercesDOMTree,
CustomerBuffer * tuxbuf, CustType_Ext * extinfo)
```

```
int (* soap_out_tuxedo__CUSTBUF) (void ** xercesDOMTree,  
CustomerBuffer * tuxbuf, CustType_Ext * extinfo)  
    .....  
}
```

The function pointer (`* soap_in_tuxedo__CUSTBUF`) points to the customized function that converts the SOAP message payload to Tuxedo typed buffer.

The function pointer (`* soap_out_tuxedo__CUSTBUF`) points to the customized function that converts the Tuxedo typed buffer to SOAP message payload.

You may implement both functions defined in the message conversion plug-in vtable structure if needed. You may also implement one function and set the other function with a NULL pointer.

How Message Conversion Plug-in Works in an Inbound Call Scenario

An inbound call scenario is an external Web service program that invokes a Tuxedo service through the Oracle SALT gateway. [Figure 5-1](#) depicts message streaming between a Web service client and a Tuxedo domain.

Figure 5-1 Message Conversion Plug-in Works in an Inbound Call Scenario



When a SOAP request message is delivered to the GWWS server, GWWS tries to find if there is a message conversion plug-in instance associated with the input message conversion of the target

service. If there is an associated instance, the GWWS invokes the customized (`*soap_in_tuxedo__CUSTBUF`) function implemented in the plug-in instance.

When a Tuxedo response buffer is returned from the Tuxedo service, GWWS tries to find if there is a message conversion plug-in instance associated with the output message conversion of the target service. If there is an associated instance, GWWS invokes the customized (`*soap_out_tuxedo__CUSTBUF`) function implemented in the plug-in instance.

How Message Conversion Plug-in Works in an Outbound Call Scenario

An outbound call scenario is a Tuxedo program that invokes an external Web service through the Oracle SALT gateway. [Figure 5-2](#) depicts message streaming between a Tuxedo domain and a Web service application.

Figure 5-2 Message Conversion Plug-in Works in an Outbound Call Scenario



When a Tuxedo request buffer is delivered to the GWWS server, GWWS tries to find if there is a message conversion plug-in instance associated with the input message conversion of the target service. If there is an associated instance, GWWS invokes the customized (`*soap_out_tuxedo__CUSTBUF`) function implemented in the plug-in instance.

When a SOAP response message is returned from the external Web service application, GWWS tries to find if there is a message conversion plug-in instance associated with the output message conversion of the target service. If there is an associated instance, GWWS invokes the customized (`*soap_in_tuxedo__CUSTBUF`) function implemented in the plug-in instance.

When Do We Need Message Conversion Plug-in

Table 5-1 lists several message conversion plug-in use cases.

Table 5-1 Message Conversion Plug-in Use Cases

	Scenario Description	soap_in_tuxedo_CUSTBUF	soap_out_tuxedo_CUSTBUF
Tuxedo Originated Service	A SOAP message payload is being transformed into a custom typed buffer	Required	N/A
	A custom typed buffer is being transformed into a SOAP message payload	N/A	Required
	A Tuxedo service input and/or output buffer is associated with a customized XML schema definition, when a SOAP message payload is being transformed into this buffer	Non XML typed buffer: Required	N/A
	A Tuxedo service input and/or output buffer is associated with a customized XML schema definition, when this buffer is being transformed into a SOAP message payload	N/A	Non XML typed buffer: Required
	All other general cases when a SOAP message payload is being transformed to a Tuxedo buffer	Optional	N/A
	All other general cases when a Tuxedo buffer is being transformed into a SOAP message payload	N/A	Optional

Table 5-1 Message Conversion Plug-in Use Cases

	Scenario Description	soap_in_tuxedo_CUSTBUF	soap_out_tuxedo_CUSTBUF
Web Service Originated Service	All cases when a Tuxedo buffer is being transformed to a SOAP message payload	N/A	Optional
	All cases when a SOAP message payload is being transformed into a Tuxedo buffer	Optional	N/A

From [Table 5-1](#), the following message conversion plug-ins general rules are applied.

- If a Tuxedo originated service consumes custom typed buffer, the message conversion plug-in is required. Tuxedo framework does not understand the detailed data structure of the custom typed buffer, therefore SALT default data type conversion rules cannot be applied.
- If the input and/or output (no matter returned with `TPSUCCESS` or `TPFAIL`) buffer of a Tuxedo originated service is associated with an external XML Schema, you should develop the message conversion plug-ins to handle the transformation manually, unless you are sure that the SALT default buffer type-based conversion rules can handle it correctly.
 - For example, if you associate your own XML Schema with a Tuxedo service FML32 typed buffer, you must provide a message conversion plug-in since SALT default data mapping routines may not understand the SOAP message payload structure when trying to convert into the FML typed buffer. Contrarily, the SOAP message payload structure converted from the FML typed buffer may be tremendously different from the XML shape defined via your own XML Schema.
 - If you associate your own XML Schema with a Tuxedo service XML typed buffer, most of time you do not have to provide a message conversion plug-in. This is because SALT just passes the XML data as is in both message conversion directions.

For more information about how to associate external XML Schema definition with the input, output and error buffer of a Tuxedo Service, see “[Defining Tuxedo Service Contract with Service Metadata Repository](#)” in the *Oracle SALT Administration Guide*.

- You can develop message conversion plug-ins for any message level conversion to replace SALT default message conversion routines as needed.

Developing a Message Conversion Plug-in Instance

Converting a SOAP Message Payload to a Tuxedo Buffer

The following function should be implemented in order to convert a SOAP XML payload to a Tuxedo buffer:

```
CustomerBuffer * (* soap_in_tuxedo__CUSTBUF) (void * xercesDOM,  
CustomerBuffer *a, CustType_Ext * extinfo);
```

Synopsis

```
#include <custtype_pi_ex.h>
```

```
CustomerBuffer * myxml2buffer (void * xercesDOM, CustomerBuffer *a,  
CustType_Ext * extinfo);
```

`myxml2buffer` is an arbitrary customized function name.

Description

The implemented function should have the capability to parse the given XML buffer and convert concrete data items to a Tuxedo custom typed buffer instance.

The input parameter, `char * xmlbuf`, indicates a NULL terminated string with the XML format data stream. Please note that the XML data is the actual XML payload for the custom typed buffer, *not* the whole SOAP envelop document or the whole SOAP body document.

The input parameter, `char * type`, indicates the custom typed buffer type name, this parameter is used to verify that the GWWS server expected custom typed buffer handler matches the current plug-in function.

The output parameter, `CustomerBuffer *a`, is used to store the allocated custom typed buffer instance. A Tuxedo custom typed buffer must be allocated by this plug-in function via the ATMI function `tpalloc()`. Plug-in code is not responsible to free the allocated custom typed buffer, it is automatically destroyed by the GWWS server if it is not used.

Diagnostics

If successful, this function must return the pointer value of input parameter `CustomerBuffer * a`.

If it fails, this function returns NULL as shown in [Listing 5-5](#).

Listing 5-5 Converting XML Effective Payload to Tuxedo Custom Typed Buffer Pseudo Code

```

CustomerBuffer * myxml2buffer (void * xercesDOM, CustomerBuffer *a,
CustType_Ext * extinfo)
{
    // casting the input void * xercesDOM to class DOMDocument object
    DOMDocument * DOMTree =

    // allocate custom typed buffer via tmalloc
    a->buf = tmalloc("MYTYPE", "MYSUBTYPE", 1024);
    a->len = 1024;

    // fetch data from DOMTree and set it into custom typed buffer
    DOMTree ==> a->buf;
    if ( error ) {
        release ( DOMTree );
        tpfree(a->buf);
        a->buf = NULL;
        a->len = 0;
        return NULL;
    }

    release ( DOMTree );

    return a;
}

```

Tip: Tuxedo bundled Xerces library can be used for XML parsing. Tuxedo 8.1 bundles Xerces 1.7 and Tuxedo 9.1 bundles Xerces 2.5

Converting a Tuxedo Buffer to a SOAP Message Payload

The following function should be implemented in order to convert a custom typed buffer to SOAP XML payload:

```
int (*soap_out_tuxedo__CUSTBUF)(char ** xmlbuf, CustomerBuffer * a, char * type);
```

Synopsis

```
#include <custtype_pi_ex.h>
```

```
int * mybuffer2xml (char ** xmlbuf, CustomerBuffer *a, char * type);
```

"mybuffer2xml" is the function name can be specified with any valid string upon your need.

Description

The implemented function has the capability to convert the given custom typed buffer instance to the single root XML document used by the SOAP message.

The input parameter, `CustomerBuffer *a`, is used to store the custom typed buffer response instance. Plug-in code is not responsible to free the allocated custom typed buffer, it is automatically destroyed by the GWWS server if it is not used.

The input parameter, `char * type`, indicates the custom typed buffer type name, this parameter can be used to verify if the SALT GWWS server expected custom typed buffer handler matches the current plug-in function.

The output parameter, `char ** xmlbuf`, is a pointer that indicates the newly converted XML payload. The XML payload buffer must be allocated by this function and use the `malloc ()` system API. Plug-in code is not responsible to free the allocated XML payload buffer, it is automatically destroyed by the GWWS server if it is not used.

Diagnostics

If successful, this function must returns 0.

If it fails, this function must return -1 as shown in [Listing 5-6](#).

Listing 5-6 Converting Tuxedo Custom Typed Buffer to SOAP XML Pseudo Code

```
int mybuffer2xml (void ** xercesDom, CustomerBuffer *a, CustType_Ext *
extinfo)
{
    // Use DOM implementation to create the xml payload
    DOMTree = CreateDOMTree( );

    if ( error )
        return -1;
```



```

// fetch data from custom typed buffer instance,
// and add data to DOMTree according to the client side needed
// XML format

a->buf ==> DOMTree;

// allocate xmlbuf buffer via malloc
* xmlbuf = malloc( expected_len(DOMTree) );
if ( error ) {
    release ( DOMTree );
    return -1;
}

// casting the DOMDocument to void * pointer and returned
DOMTree >> (* xmlbuf);
if ( error ) {
    release ( DOMTree );
    free ( (* xmlbuf) );
    return -1;
}

return 0;
}

```

WARNING: GWWS framework is responsible to release the DOMDocument created inside the plug-in function. To avoid double release, programmers must pay attention to the following Xerces API usage:

If the DOMDocument is constructed from an XML string through `XercesDOMParser::parse()` API. You must use `XercesDOMParser::adoptDocument()` to get the pointer of the DOMDocument object. You must not use `XercesDOMParser::getDocument()` to get the pointer of the DOMDocument object because the DOMDocument object is maintained by the XercesDOMParser object and is released when deleting the XercesDOMParser object if you do not de-couple the

DOMDocument from the XercesDOMParser via the
`XercesDOMParser::getDocument()` function.

SALT 1.1 Custom Buffer Type Conversion Plug-in Compatibility

SALT 1.1 Custom Buffer Type Conversion Plug-in provides the customized message conversion mechanism only for Tuxedo custom buffer types.

[Table 5-2](#) compares the SALT Message Conversion Plug-in and the SALT 1.1 Custom Buffer Type Conversion Plug-in.

Table 5-2 SALT 10gR3 Message Conversion Plug-in / SALT 1.1 Custom Buffer Type Conversion Plug-in Comparison

SALT 1.1 Custom Buffer Type Plug-in	SALT 10gR3 Message Conversion Plug-in
Plug-in ID is "P_CUSTOM_TYPE"	Plug-in ID is "P_CUSTOM_TYPE"
Plug-in Name must be the same as the supported custom buffer type name	Plug-in Name can be any meaningful value, which is only used to distinguish from other plug-in instances.
Only supports message conversion between SOAP message payload and Tuxedo custom buffer types	Supports message conversion between SOAP message payload and any kind of Tuxedo buffer type
Buffer type level association. Each plug-in instance must be named the same as the supported custom buffer type name. Each custom buffer type can only have one plug-in implementation. One custom buffer type can associate with a plug-in instance, and used by all the services	Message level association. Each Tuxedo service can associate plug-in instances with its input and/or output buffers respectively through the plug-in instance name.
SOAP message payload is saved as a NULL terminated string for plug-in programming	SOAP message payload is saved as a Xerces DOM Document for plug-in programming

Please note that the SALT 1.1 Custom Buffer Type Plug-in shared library cannot be used directly in SALT 10gR3. You must perform the following tasks to upgrade it to a SALT 10gR3 message conversion plug-in:

1. Re-implement function (`*soap_in_tuxedo__CUSTBUF`) and (`*soap_out_tuxedo__CUSTBUF`) according to new SALT 10gR3 message conversion plug-in `vtable` function prototype API. The major change is that SOAP message payload is saved as an Xerces class `DOMDocument` object instead of the old string value.
2. Re-compile your functions as the shared library and configure this shared library in the SALT Deployment file so that it can be loaded by GWWS servers.

Tip: You do not have to manually associate the upgraded message conversion plug-ins with service buffers. If a custom typed buffer is involved in the message conversion at runtime, GWWS can automatically search a message conversion plug-in that has the same name as the buffer type name if no explicit message conversion plug-in interface is configured.

Programming Outbound Authentication Plug-Ins

When a Tuxedo client accesses Web services via SOAP/HTTP, the client may be required to send a username and password to the server to perform HTTP Basic Authentication. The Tuxedo clients uses `tpinit()` to send a username and password when registering to the Tuxedo domain. However, this username is used by Tuxedo and is not the same as the one used by the Web service (the password may be different as well).

To map the usernames, Oracle SALT provides a plug-in interface (Credential-Mapping Interface) that allows you to choose which username and password is sent to the Web service.

How Outbound Authentication Plug-Ins Work

When a Tuxedo client calls a Web service, it actually calls the GWWS server that declares the Web service as a Tuxedo service. The user id and group id (defined in `tpusr` and `tpgrp` files) are sent to the GWWS. The GWWS then checks whether the Web service has a configuration item `<Realm>`. If it does, the GWWS:

tries to invoke the `vtable gwws_pi_map_http_basic` function to map the Tuxedo userid into the username and password for the HTTP Realm of the server.

- for successful calls, encodes the returned username and password with `Base64` and sends it in the HTTP header field “Authorization: Basic” if the call is successful
- for failed calls, returns a failure to the Tuxedo Client without invoking the Web service.

Implementing a Credential Mapping Interface Plug-In

Using the following scenario:

- An existing Web service, *myservice*, sited on `http://www.abc.com/webservice`, requires HTTP Basic Authentication. The username is “test”, the password is “1234,” and the realm is “myrealm”.
- After converting the Web service WSDL into the SALT configuration file (using `wSDL2CVT`), add the `<Realm>myrealm</Ream>` element to the endpoint definition in the WSDL file.

Perform the following steps to implement a Oracle SALT plug-in interface:

1. Write the functions to map the “myrealm” Tuxedo UID/GID to username/password on `www.abc.com`.

- Use `Credmap_HTTP_Basic()`;

This function is used to return the HTTP username/password. The function prototype defined in `credmap_pi_ex.h`

2. Write the following three plug-in register functions. For more information, see [Plug-In Register Functions](#).

- `_ws_pi_init_P_CREDENMAP_TEST(char * params, void ** priv_ptr);`

This function is invoked when the GWWS server attempts to load the plug-in shared library during startup.

- `_ws_pi_exit_P_CREDENMAP_TEST(void * priv);`

This function is invoked when the GWWS server unloads the plug-in shared library during the shutdown phase.

- `_ws_pi_set_vtbl_P_CREDENMAP_TEST(void * vtbl);`

Set the `gwws_pi_map_http_basic` entry in vtable structure `credmap_vtable` with the `Credmap_HTTP_Basic()` function implemented in step 1.

3. You can also write the optional function

- `_ws_pi_get_Id_and_Names(int * params, char ** ids, char ** names);`

This function is invoked when the GWWS server attempts to load the plug-in shared library during startup to determine what library interfaces are implemented. For more information, see [Plug-In Register Functions](#).

4. Compile the previous four or five functions into one shared library, `credmap_plugin.so`.
 5. Configure the plug-in interface in the SALT deployment file.
- Configure the plug-in interface as shown in [Listing 5-7](#).

Listing 5-7 Custom Typed Buffer Plug-In Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<Deployment xmlns="http://www.bea.com/Tuxedo/SALTDEPLOY/2007">
    . . . . .
    . . . . .
    <System>
        <Plugin>
            <Interface
                id="P_CREDENMAP"
                name="TEST"
                library="credmap_plugin.dll" />
        </Plugin>
    </System>
</Deployment>
```

Mapping the Tuxedo UID and HTTP Username

The following function should be implemented in order to return username/password for HTTP Basic Authentication:

```
typedef int (* GWWS_PI_CREDMAP_PASSTEXT) (char * domain, char * realm, char
* t_userid, char * t_grpid, Cred_UserPass * credential);
```

Synopsis

```
#include <credmap_pi_ex.h>
typedef struct Cred_UserPass_s {
    char username[UP_USERNAME_LEN];
    char password[UP_PASSWORD_LEN];
} Cred_UserPass;

int gwws_pi_map_http_basic (char * domain, char * realm, char * t_uid, char
* t_gid, Cred_UserPass * credential);
```

The "gwws_pi_map_http_basic" function name can be specified with any valid string as needed.

Description

The implemented function has the capability to determine authorization credentials (usernames and passwords) used for authorizing users with a given Tuxedo uid and gid for a given domain and realm.

The input parameters, `char * domain` and `char * realm`, represent the domain name and HTTP Realm that the Web service belongs to. The plug-in code must use them to determine the scope to find appropriate credentials.

The input parameters, `char * t_uid` and `char * t_gid`, are strings that contain Tuxedo user ID and group ID number values respectively. These two parameters may be used to find the username.

The output parameter, `Cred_UserPass * credential`, is a pointer that indicates a pre-allocated buffer storing the returned username/password. The plug-in code is not responsible to allocate the buffer.

Notes: Tuxedo user ID is available only when `*SECURITY` is set as `USER_AUTH` or higher in the `UBBCONFIG` file. Group ID is available when `*SECURITY` is set as `ACL` or higher. The default is "0".

Diagnostics

If successful, this function returns 0. If it fails, it returns -1 as shown in [Listing 5-8](#).

Listing 5-8 Credential Mapping for HTTP Basic Authentication Pseudo Code

```
int Credmap_HTTP_Basic(char * domain, char * realm, char * t_uid, char *
t_gid, Cred_UserPass * credential)
{
    // Use domain and realm to determine scope
    credentialList = FindAllCredentialForDomainAndRealm(domain, realm);

    if ( error happens )
        return -1;

    // find appropriate credential in the scope
```

```
foreach cred in credentialList {
    if (t_uid and t_gid match) {
        *credential = cred;
        return 0;
    }
}
if ( not found and no default credential) {
    return -1;
}

*credential = default_credential;
return 0;
}
```

Tip: The credentials can be stored in the database with domain and realm as the key or index.

Oracle SALT SCA Programming

This chapter contains the following topics:

- [Overview](#)
- [SCA Utilities](#)
- [SCA Client Programming](#)
- [SCA Component Programming](#)
- [SCA Python and Ruby Programming](#)
- [SCA Structure Support](#)
- [SCA Remote Protocol Support](#)
- [SCA Binding](#)
- [SCA Data Type Mapping](#)
- [SCA and Tuxedo Interoperability](#)
- [SCA Transactions](#)
- [SCA Security](#)

Overview

One important aspect of Service Component Architecture (SCA) is the introduction of a new programming model. As part of the Tuxedo architecture, SCA allows you to better blend high-output, high-availability and scalable applications in an SOA environment.

SCA components run on top of the Tuxedo infrastructure using ATMI binding. The ATMI binding implementation provides native Tuxedo communications between SCA components, as well as SCA components and Tuxedo programs (clients and servers).

In addition to the programming model, the Service Component Definition Language (SCDL) describes what components can perform in terms of interactions between each other, and instructs the framework to set-up necessary links (wires).

SCA Utilities

The following utilities are used in conjunction with Oracle SALT SCA programming:

- `buildscaclient`: Builds client processes that call SCA components.
- `buildscacomponent`: Builds SCA components.
- `buildscaserver`: Builds an Oracle Tuxedo server containing SCA components.
- `SCAHOST`: Generic server for Python or Ruby SCA components.
- `scatuxgen`: Generates Tuxedo Service Metadata Repository interface information from an SCA interface.
- `scastructc32`, `scastructc`: Structure description file compiler.
- `scastructdis32`, `scastructdis`: Binary structure and view files disassembler.
- `tuxscagen`: Generates SCA, SCDL, and server-side interface files for Tuxedo services.

For more information, see the [Oracle SALT Command Reference](#).

SCA Client Programming

The runtime reference binding extension is the implementation of the client-side aspect of the SCA container. It encapsulates the necessary code used to call other services, SCA components, Tuxedo servers or even Web services, transparently from an SCA-based component.

SCA Client Programming Steps

Developing SCA client programs requires the following steps:

1. [Setting Up the Client Directory Structure](#)
2. [Developing the Client Application](#)
3. [Composing the SCDL Descriptor](#)
4. [Building the Client Application](#)
5. [Running the Client Application](#)
6. [Handling TPFAIL Data](#)

Setting Up the Client Directory Structure

You must define the applications physical representation. Strict SCA client applications are SCA component types. [Listing 6-1](#) shows the directory structure used to place SCA components in an application.

Listing 6-1 SCA Component Directory Structure

```
myApplication/ (top-level directory, designated by the APPDIR environment
variable)
    root.composite (SCDL top-level composite, contains the list of
components in this application)
    myClient/ (directory containing actual client component described in
this section)
        myClient.composite (SCDL for the client component)
        myClient.cpp (client program source file)
        TuxService.h (interface of component called by client program)
```

[Listing 6-2](#) shows an example of typical `root.composite` content.

Listing 6-2 root.composite Content

```

<composite xmlns="http://www.oxa.org/xmlns/sca/1.0"
  name="simple.app">
  <component name="myClientComponent">
    <implementation.composite name="myClient"/>
  </component>
</composite>

```

The `implementation.composite@name` parameter references the directory that contains the component named 'myClientComponent'. This value is required at runtime. For more information, see [Running the Client Application](#).

Developing the Client Application

Client programs are required to implement a call to a single API. This following call is required in order to set up the SCA runtime:

```

...
    CompositeContext theContext = CompositeContext::getCurrent();
...

```

Actual calls are based on an interface. This interface is usually developed along with the component being called. In the case of existing Tuxedo ATMI services, this interface can be generated by accessing the Tuxedo METADATA repository. For more information, see the [Oracle SALT Administration Guide](#) and `tuxscagen`, `scatuxgen` in the Oracle SALT Reference Guide.

In the case of calling external Web services, an interface matching the service WSDL must be provided. For more information, see [SCA Data Type Mapping](#) for the correspondence between WSDL types and C++ types.

[Listing 6-3](#) shows an interface example.

Listing 6-3 Interface Example

```

#include <string>
/**
    * Tuxedo service business interface

```

```

*/
class TuxService
{
public:
virtual std::string TOUPPER(const std::string inputString) = 0;
};

```

In the interface shown in [Listing 6-3](#), a single method `TOUPPER` is defined. It takes a single parameter of type `std::string`, and returns a value of type `std::string`. This interface needs to be located in its own `.h` file, and is referenced by the client program by including the `.h` file.

[Listing 6-4](#) shows an example of a succession of calls required to perform an invocation.

Listing 6-4 Invocation Call Example

```

// SCA definitions
// These also include a Tuxedo-specific exception definition:
ATMIBindingException
#include "tuxsca.h"
// Include interface
#include "TuxService.h"
...
// A client program uses the CompositeContext class
CompositeContext theContext = CompositeContext::getCurrent();
...
// Locate Service
TuxService* toupperService =
    (TuxService *)theContext.locateService("TOUPPER");
...
// Perform invocation
const std::string result = toupperService->TOUPPER("somestring");
...

```

Notes: The invocation itself is equivalent to making a local call (as if the class were in another file linked in the program itself).

For detailed code examples, see the SCA samples located in following directories:

- UNIX samples: `$TUXDIR/samples/salt/sca`
- Windows samples: `%TUXDIR%\samples\salt\sca`

Composing the SCDL Descriptor

The link between the local call and the actual component is made by defining a binding in the SCDL side-file. For example, [Listing 6-4](#) shows a call to an existing Tuxedo ATMI service, the SCDL descriptor shown in [Listing 6-5](#) should be used. This SCDL is contained in a file called `<componentname>.composite`.

Listing 6-5 SCDL Descriptor

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  name="simpapp.client">
  <reference name="TOUPPER">
    <interface.cpp header="TuxService.h"/>
    <binding.atmi requires="legacy">
      <inputBufferType target="TOUPPER">STRING</inputBufferType>
      <outputBufferType target="TOUPPER">STRING</outputBufferType>
    </binding.atmi>
  </reference>
</composite>
```

This composite file indicates that a client component may perform a call to the `TOUPPER` reference, and that this call is performed using the ATMI binding. In effect, this results in a `tpcall()` to the "TOUPPER" Tuxedo service. This Tuxedo service may be an actual existing Tuxedo ATMI service, or another SCA component exposed using the ATMI binding. For more information, see [SCA Component Programming](#).

The `inputBufferType` and `outputBufferType` elements are used to determine the type of Tuxedo buffer used to exchange data. For more information, see [SCA Data Type Mapping](#) and the [ATMI Binding Element Reference](#) for a description of all possible values that can be used in the `binding.atmi` element.

Building the Client Application

Once all the elements are in place, the client program is built using the `buildscaclient` command. You must do the following steps:

1. Navigate to the directory containing the client source and SCDL composite files
2. Execute the following command:

```
$ buildscaclient -c myClientComponent -s . -f myClient.cpp
```

This command verifies the SCDL code, and builds the following required elements:

- A shared library (or DLL on Windows) containing generated proxy code
- The client program itself

If no syntax or compilation error is found, the client program is ready to use.

Running the Client Application

To execute the client program, the following environment variables are required:

- `APPDIR` - designates the application directory; in the case of SCA this typically contains the top-level SCDL composite.
- `SCA_COMPONENT` - the default SCA component (the value 'myClientComponent' in the example shown in [Listing 6-2](#)). It tells the SCA runtime where to start when looking for services in the `locateService()` call.

Invoking Existing Tuxedo Services

Access to existing Tuxedo ATMI services from an SCA client program can be simplified using the examples shown in [Listing 6-6](#), [Listing 6-7](#), and [Listing 6-8](#).

Note: These examples can also be used for server-side SCA components.

Starting from a Tuxedo METADATA repository entry as shown in [Listing 6-6](#), the `tuxscagen` command can be used to generate interface and SCDL.

Listing 6-6 SCA Components Calling an Existing Tuxedo Service

```
service=TestString
tuxservice=ECHO
servicetype=service
```

```
inbuf=STRING
outbuf=STRING

service=TestCarray
tuxservice=ECHO
servicetype=service
inbuf=CARRAY
outbuf=CARRAY
```

Listing 6-7 Generated Header

```
#ifndef ECHO_h
#define ECHO_h
#include <string>
#include <tuxsca.h>
class ECHO
{
public:
    virtual std::string TestString(const std::string arg) = 0;
    virtual std::string TestCarray(const struct carray_t * arg) = 0; };
#endif /* ECHO_h */
```

Listing 6-8 Generated SCDL Reference

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osea.org/xmlns/sca/1.0"
name="ECHO">
    <reference name="ECHO">
        <interface.cpp header="ECHO.h" />
        <binding.atmi requires="legacy">
            <serviceType target="TestString">RequestResponse</serviceType>
            <inputBufferType target="TestString">STRING</inputBufferType>
            <outputBufferType target="TestString">STRING</outputBufferType>
            <serviceType target="TestCarray">RequestResponse</serviceType>
            <inputBufferType target="TestCarray">CARRAY</inputBufferType>
            <outputBufferType target="TestCarray">CARRAY</outputBufferType>
```



```

    </binding.atmi>
  </reference>
</composite>

```

The steps used to invoke these services are identical to the examples shown in [Listing 6-6](#) through [Listing 6-8](#).

Handling TPFALL Data

Invoking a non-SCA Tuxedo ATMI service may return an error, but still send back data by using `tpreturn(TPFALL, ...)`. When this happens, an SCA client or component is interrupted by the `ATMIBindingException` type.

The data returned by the service, if present, can be obtained by using the `ATMIBindingException.getData()` API. For more information see, [TPFALL Return Data](#).

The example in [Listing 6-9](#) corresponds to a `binding.atmi` definition as shown in [Listing 6-10](#).

Listing 6-9 Invocation Interruption Example

```

...
    try {
        const char* result = toupperService->charToup("someInput");
    } catch (tuscany::sca::atmi::ATMIBindingException& abe) {
        // Returns a pointer to data corresponding to
        // mapping defined in <errorBufferType> element
        // in SCDL
        const char* *result = (const char **)abe.getData();
        if (abe.getData() == NULL) {
            // No data was returned
        } else {
            // Process data returned
            ...
        }
    } catch (tuscany::sca::ServiceInvocationException& sie) {
        // Other type of exception is returned
    }

```

```
    }  
    ...
```

Listing 6-10 /binding.atmi Definition

```
...  
    <binding.atmi requires="legacy">  
        <inputBufferType target="charToup">STRING</inputBufferType>  
        <outputBufferType  
target="charToup">STRING</outputBufferType>  
        <errorBufferType target="charToup">STRING</errorBufferType>  
</binding.atmi/>  
...
```

Other returned data types must be cast to the corresponding type. For example, an invocation returning a `commonj::sdo::DataObjectPtr` as shown in [Listing 6-11](#).

Listing 6-11 SCDL Invocation Example

```
...  
    <errorBufferType target="myMethod">FML32/myType</errorBufferType>  
...
```

The `ATMIBindingException.getData()` result is shown in [Listing 6-12](#).

Listing 6-12 ATMIBindingException.getData() Results

```
...  
        catch (tuscanysca.atmi.ATMIBindingException& abe) {  
            const commonj::sdo::DataObjectPtr *result =  
                (const commonj::sdo::DataObjectPtr *)abe.getData();  
...
```

The rules for returning TPFAIL data to the calling application are as follows:

- For each `<errorBufferType>`, a canonical type is defined, where `<errorBufferType>` is converted. When the `<errorBufferType>` is equal to the `<outputBufferType>`, the canonical type is the same C++ type that is returned in a successful service implementation.
- When the `<errorBufferType>` is different from the `<outputBufferType>`, the canonical type is as follows:
 - For STRING buffers, a C++ `char*` or `char[]` data type.
 - For MBSTRING buffers, a C++ `wchar_t*` or `wchar_t[]`.
 - For CARRAY buffers, a C++ `CARRAY_PTR`.
 - For X_OCTET buffers, a C++ `X_OCTET_PTR`.
 - For XML buffers, a C++ `XML_PTR`.
 - For FML, FML32, VIEW, VIEW32, X_COMMON, and X_C_TYPE buffers, a C++ `commonj::sdo::DataObjectPtr`.
- In each case, the value returned by `getData()` is a pointer to one of the types listed above.

For more conversion rules between Tuxedo buffer types and C++ data information, see [SCA Data Type Mapping](#).

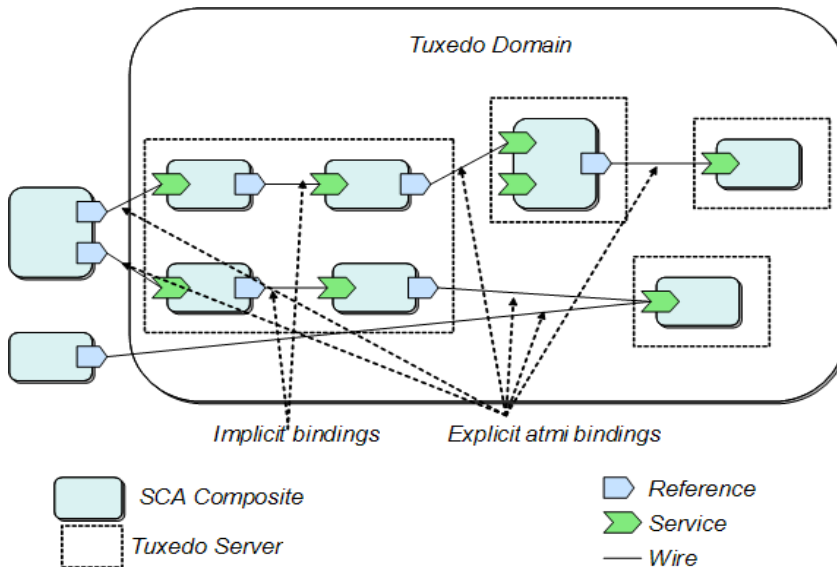
SCA Component Programming

The SCA Component terminology designates SCA runtime artifacts that can be invoked by other SCA or non-SCA runtime components. In turn, these SCA Components can perform calls to other SCA or non-SCA components. This is different from strict SCA clients which can only make calls to other SCA or non-SCA components, but cannot be invoked.

The Oracle SALT SCA container provides the capability of hosting SCA components in an Oracle Tuxedo server environment. This allows you to take full advantage of proven Oracle Tuxedo qualities: *reliability*, *scalability* and *performance*.

[Figure 6-1](#) summarizes SCA components and Tuxedo server mapping rules.

Figure 6-1 SCA Component and Tuxedo Server Mapping Rules



While SCA components using Tuxedo references do not require special processing, SCA components offering services must still be handled in a Tuxedo environment.

The mapping is as follows:

- An SCA composite declaring one or more services with a `<binding.atmi>` definition maps to a single Tuxedo server advertising the same number of services as the SCA composite.
- There can be more than one composite.
- Composites can be nested.
- Promotion handling:
 - A composite promoting a service contained in a nested component results in the promoted service being advertised as a Tuxedo service.
 - A service declared in a component, but not promoted, is not advertised.
- The resulting Tuxedo server advertises as many services as there are `binding.atmi` sections in the SCDL definition

- Interfaces may declare multiple methods. Each method is linked to a Tuxedo native service using the `/binding.atmi/@map` attribute. A method not declared via the `/binding.atmi/@map` attribute is not accessible through Tuxedo. The use of duplicate service names are detected at server generation time, so that Tuxedo service names-to-interface method mapping in a single Tuxedo server instance is 1:1.
- A generated Tuxedo server acts as a proxy for SCA components. An instance of this generated server corresponds to an SCA composite as defined in the SCDL configuration. Such servers are deployed as necessary by the Tuxedo administrator.

SCA composites are deployed in a Tuxedo application by configuring instances of generated SCA servers in the `UBBCONFIG` file. Multiple instances are allowed. Multi-threading capabilities are also allowed and controllable using already-existing Tuxedo features.

SCA Component Programming Steps

The steps required for developing SCA component programs are as follows:

1. [Setting Up the Component Directory](#)
2. [Developing the Component Implementation](#)
3. [Composing the SCDL Descriptor](#)
4. [Compiling and Linking the Components](#)
5. [Building the Tuxedo Server Host](#)

Setting Up the Component Directory

You must first define the applications physical representation. [Listing 6-13](#) shows the directory structure used to place SCA components in an application:

Listing 6-13 SCA Component Directory Structure

```
myApplication/ (top-level directory, designated by the APPDIR environment
variable)
    root.composite (SCDL top-level composite, contains the list of
components in this application)
    myComponent/ (directory containing actual component described in this
section)
        myComponent.composite (SCDL for the component)
```

```
myComponent.componentType  
myComponentImpl.cpp (component implementation source file)  
TuxService.h (interface of component being exposed)  
TuxServiceImpl.h (component implementation definitions)
```

[Listing 6-14](#) shows typical `root.composite` content.

Listing 6-14 `root.composite` Content

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"  
  name="simple.app">  
  <component name="myComponent">  
    <implementation.composite name="myComponent"/>  
  </component>  
</composite>
```

The `implementation.composite@name` parameter references the directory that contains the 'myComponent' component.

Developing the Component Implementation

Components designed to be called by other components do not need to be aware of the SCA runtime. There are, however, limitations in terms of interface capabilities, such as:

- C++ classes (other than `std::string` and `commonj::sdo::DataObjectPtr`) cannot be used as parameters or return values
- Parameter arrays are not supported

For more information, see [SCA Data Type Mapping](#).

[Listing 6-15](#) shows an example of an interface implemented for a client program.

Listing 6-15 Component Implementation Interface

```
#include <string>
/**
 * Tuxedo service business interface
 */
class TuxService
{
public:
    virtual std::string TOUPPER(const std::string inputString) = 0;
};
```

The component implementation then generally consists of two source files (as shown [Listing 6-16](#) and [Listing 6-17](#) respectively):

- component implementation definitions, contained in a <servicename>Impl.h file, and
- component implementation, contained in a <servicename>Impl.cpp file

Listing 6-16 Example (TuxServiceImpl.h):

```
#include "TuxService.h"

/**
 * TuxServiceImpl component implementation class
 */
class TuxServiceImpl: public TuxService
{
public:
    virtual std::string toupper(const std::string inputString);
};
```

Listing 6-17 Example (TuxServiceImpl.cpp):

```
#include "TuxServiceImpl.h"
#include "tuxsca.h"

using namespace std;
using namespace osoa::sca;

/**
 * TuxServiceImpl component implementation
 */
std::string TuxServiceImpl::toupper(const string inputString)
{
    string result = inputString;

    int len = inputString.size();

    for (int i = 0; i < len; i++) {
        result[i] = std::toupper(inputString[i]);
    }

    return result;
}
```

Additionally, a side-file (`componentType`), is required. It contains the necessary information for the SCA wrapper generation and possibly proxy code (if this component calls another component).

This `componentType` file (`<componentname>Impl.componentType`) is an SCDL file type. [Listing 6-18](#) shows an example of a `componentType` file (`TuxServiceImpl.componentType`).

Listing 6-18 componentType File Example

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0" >
    <service name="TuxService">
```



```

        <interface.cpp header="TuxService.h"/>
    </service>
</componentType>

```

Composing the SCDL Descriptor

The link between the local implementation and the actual component is made by defining a binding in the SCDL side-file. For example, for the file type in [Listing 6-18](#) to be exposed as a Tuxedo ATMI service, the SCDL in [Listing 6-19](#) should be used. This SCDL is contained in a file called `<componentname>.composite` (for example, `myComponent.composite`).

Listing 6-19 Example SCDL Descriptor

```

<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="myComponent">
    <service name="TuxService">
        <interface.cpp header="TuxService.h"/>
        <binding.atmi requires="legacy"/>
        <map target="toupper">TUXSVC</map>
        <inputBufferType target="toupper">STRING</inputBufferType>
        <outputBufferType target="toupper">STRING</outputBufferType>
        <reference>MYComponent</reference>
    </service>
    <component name="MYComponent">
        <implementation.cpp library="TuxService" header="TuxServiceImpl.h"/>
    </component>
</composite>

```

This composite file indicates that the service, `mySVC`, can be invoked via the Tuxedo infrastructure. It further indicates that the `toupper()` method is advertised as the `TUXSVC` service in the Oracle Tuxedo system. Once initialized, another SCA component may now call this service, as well as a non-SCA Tuxedo ATMI client.

The `inputBufferType` and `outputBufferType` elements are used to determine the type of Tuxedo buffer used to exchange data. For more information, see [SCA Data Type Mapping](#) and the [ATMI Binding Element Reference](#) for a description of all possible values that can be used in the `binding.atmi` element.

Note: The `mycomponent.componentType` service name should be same as the composite file, otherwise an exception is thrown.

Compiling and Linking the Components

Once all the elements are in place, the component is built using the `buildscacomponent` command. The steps are as follows:

1. Navigate to the `APPDIR` directory. The component and side files should be in its own directory one level down
2. Execute the following command:

```
$ buildscacomponent -c myComponent -s . -f TuxServiceImpl.cpp
```

This command verifies the SCDL code, and builds the following required elements:

- A shared library (or DLL on Windows) containing generated proxy code

Building the Tuxedo Server Host

In order for components to be supported in a Tuxedo environment, a host Tuxedo server must be built. This is achieved using the `buildscaserver` command.

For example: `$ buildscaserver -c myComponent -s . -o mySCAServer`

When the command is executed, `mySCAServer` is ready to be used. It automatically locates the component(s) to be deployed according to the SCDL, and performs the appropriate Tuxedo/SCA associations.

SCA Python and Ruby Programming

This section contains the following topics:

- [SCA Python and Ruby Programming Overview](#)

- [Python and Ruby Client Programming](#)
- [Python and Ruby Component Programming](#)
- [Python and Ruby Data Type Mapping](#)
- [Python and Ruby Binding](#)

SCA Python and Ruby Programming Overview

Integration of Python or Ruby scripts in an environment such as Tuxedo via the SALT, is intended for providing additional flexibility in terms of program development.

Python and Ruby are comparable object-oriented scripting languages that offer many advantages over C/C++:

- No compilation
- Dynamic data typing
- Garbage collection
- Existing libraries of utility functions and objects

SALT SCA Python and Ruby support provides a set of APIs to perform SCA calls from Python or Ruby client programs, and language extensions to call Python or Ruby components. For more information, see [Python and Ruby Client Programming](#) and [Python and Ruby Component Programming](#).

The `buildscaclient`, `buildscaserver` and `buildscacomponent` commands do not need adapting for use with Python or Ruby programs, as they are not be required to produce executables or component libraries.

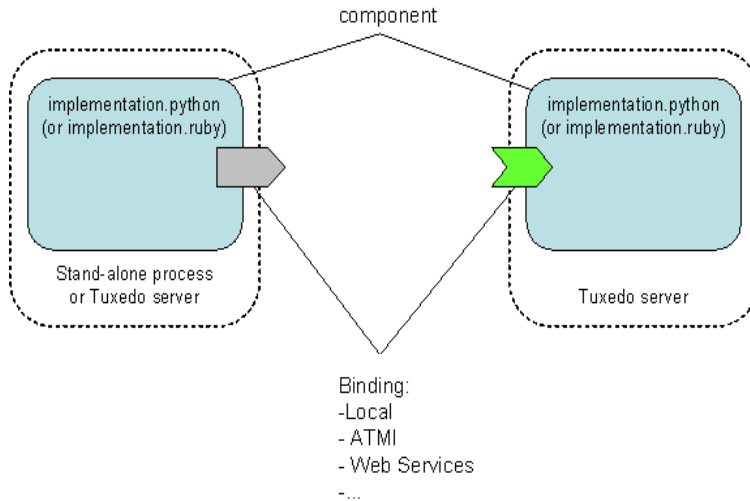
Note: A system server, `SCAHOST`, is provided to correctly marshal requests and responses to and from Python or Ruby scripts. It contains Python and Ruby scripts exposed as SCA services (via the Oracle Tuxedo Metadata Repository). The definitions describe the parameters and return types of the corresponding exposed Python or Ruby functions.

For more information, see [Python and Ruby Data Type Mapping](#) for Service Metadata Repository entry examples.

Available bindings are used from Python or Ruby programs, or are used to invoke Python or Ruby components. Like C++, the Python and Ruby language extension is binding-independent.

[Figure 6-2](#) provides an overview of the SALT SCA environment Python and Ruby support architecture.

Figure 6-2 SALT SCA Python and Ruby Programming Support Architecture



Python and Ruby Client Programming

- [SCDL Clients](#)
- [Python Clients](#)
- [Ruby Clients](#)

SCDL Clients

From a client component perspective, the SCDL code only has to mention the referenced component and possibly the binding used (that is, no interface element is required).

For example, the following snippet allows a Python or Ruby client to make an invocation to an SCA component via ATMI binding, and using the default buffer type (STRING input, STRING output):

```
<reference name="CalculatorComponent">  
    <binding.atmi/>  
</reference>
```

Python Clients

To invoke an SCA component from a Python program, you must do the following:

1. Import the SCA library using the following command:

```
import sca
```

2. Use the following API to locate the service:

```
calc = sca.locateservice("CalculatorComponent")
```

The `calc` object is used to invoke the “add” operation (for example, `result = calc.add(val1, val2)`).

Ruby Clients

To invoke an SCA component from a Ruby program, you must do the following:

1. Load the Ruby proxy extension:

```
require("sca_ruby")
```

2. Use the following API to locate the service:

```
calculator = SCA::locateService("CalculatorComponent")
```

The `calculator` object is used to invoke the “add” operation (for example, `x = calculator.add(3, 2)`).

Python and Ruby Component Programming

- [SCDL Components](#)
- [Python Components](#)
- [Ruby Components](#)

SCDL Components

In order to use Python or Ruby scripts in SCA as components, you must use the `implementation.python` and `implementation.ruby` parameters.

Note: `implementation.python` and `implementation.ruby` usage is similar to the `implementation.cpp` element (see [Listing 6-19](#) and [Listing 6-29](#)); the difference is that the `interface.python` and `interface.ruby` elements, or `.componentType` are not required.

Their syntax and attributes are as follows:

- `implementation.python`

```
<implementation.python
  module="string"
  scope="scope"? >
</implementation.python/>
```

The `implementation.python` element has the following attributes:

- `module: string(1..1)`

Name of the Python module (`.py` file) containing the operation(s) that this component offers in the form of module-level function(s).

- `scope: PythonImplementationScope(0..1)`

Identifies the scope of the component implementation. The default is `stateless`, indicating that there is no correlation between implementation instances used to dispatch service requests. A composite value indicates that all service requests are dispatched to the same implementation instance for the lifetime of the containing composite.

- `implementation.ruby`

```
<implementation.ruby
  script="string"
  class="string"
  scope="scope"? >
</implementation.ruby/>
```

The `implementation.ruby` element has the following attributes:

- `script: string(1..1)`

Name of the Ruby script (`.rb` file) containing the operation(s) that the component offers in the form of methods of a class contained in the script file. The name of the script is its full name (that is, it also includes the `.rb` extension).

- `class: string(1..1)`

Name of the Ruby class (`.rb` file) containing the operation(s) that the component offers.

- `scope: RubyImplementationScope(0..1)`

Identifies the scope of the component implementation. The default is `stateless`, indicating that there is no correlation between implementation instances used to dispatch service requests. A composite value indicates that all service requests are

dispatched to the same implementation instance for the lifetime of the containing composite.

[Listing 6-20](#) shows an example of a Python component in an SCA composite accessible using the ATMI binding. In this example, runtime looks for a Python component located in a file named `ToupperService.py` in the same location as the composite file.

Similarly, a Ruby component is required in a file named `ToupperService.rb`, in the same location as the composite file.

Listing 6-20 Python Component in an SCA Composite

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

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           name="simpapp.server">

    <service name="SCASVC">
        <binding.atmi/>
        <reference>ToupperServiceComponent</reference>
    </service>

    <component name="ToupperServiceComponent">
        <implementation.python module="ToupperService"
                               scope="composite"/>
    </component>

</composite>
```

Python Components

Python operations are exposed as module-level functions contained in a Python module file. For example, a `ToupperService.py` file would contain the code shown in [Listing 6-21](#).

Listing 6-21 Python Module File

```
def charToup(vall):  
    print "input: " + vall  
    result = "result"  
    print "Python - toupper"  
    return result
```

Parameter and return values types are dynamically determined at runtime. Application exceptions are caught by the extension runtime and re-thrown as `tuscany::sca::ServiceInvocationException`.

During input, unsupported types or an error processing an input `DataObject` results in the following exception:

```
a tuscany::sca::ServiceDataException.
```

During output, simple return types are always processed. An error generating a `DataObject` (from XML data) results in the following exception:

```
tuscany::sca::ServiceDataException.
```

For more information, see [Python and Ruby Data Type Mapping](#).

Ruby Components

Ruby operations are exposed as methods of an implementation class contained in a Ruby script file (`.rb` extension). For example, a `ToupperService.rb` file would contain the code shown in [Listing 6-22](#).

Listing 6-22 Ruby Script File

```
class ToupperService  
  
    def initialize()  
        print "Ruby - ToupperService.initialize\n"  
    end  
  
    def charToup(arg1)  
        print "Ruby - ToupperService.div\n"
```



```

    arg1.ToUpper()
end

end

```

Parameter and return values types are dynamically determined at runtime. Application exceptions are caught by the extension runtime and re-thrown as `tuscany::sca::ServiceInvocationException`.

During input, unsupported types or an error processing an input `DataObject` results in the following exception:

```
a tuscany::sca::ServiceDataException.
```

During output, simple return types are always processed. An error generating a `DataObject` (from XML data) results in the following exception: `tuscany::sca::ServiceDataException`.

For more information, see [Python and Ruby Data Type Mapping](#).

SCA Structure Support

This section contains the following topics:

- [SCA Structure Support Overview](#)
- [Using SCA Structure Description Files](#)
- [Using tuxscagen to Generate Structures](#)

Note: This section applies to application defined structures only. For information on Oracle SALT SCA defined structures, see [SCA Data Type Mapping](#).

SCA Structure Support Overview

SCA Structure support provides:

- Additional C++ structure functionality
- Improved performance for applications processing data that can be placed in a structure without significant wasted space

You must use the `struct` data type specified in the SCA method parameter definition or in the definition of a return value from an SCA method as follows:

- `struct structurename *`
- `struct structurename &`

Elements within the structure can be any of the following simple data types/arrays that are supported as an SCA parameter:

- `bool`
- `char`, `unsigned char`, `signed char`
- `wchar_t`
- `short`, `unsigned short`
- `int`, `unsigned int`
- `long`, `unsigned long`
- `long long`, `unsigned long long`
- `float`
- `double`
- `long double`
- `struct nestedstructurename`
- `typedef`

Note: The `scagen` utility parses `typedef` and `struct` keywords. For more information, see the [Oracle SALT Command Reference Guide](#).

SCA Structure Limitations

- The following cannot be specified as part of a structure”
 - `DataObjectPtr`
 - Point data types
 - `std::string` or a `std::wstring`
 - A union
 - `struct carray_t`, `struct_x_octet_t`, or `struct xml_t`
- CARRAY data is supported in the same way that it is supported for views
- `.h` and `.cpp` files referencing the use of structures are required to include a definition for the structure being used and for any nested structures contained within that structure.

Using SCA Structure Description Files

A structure description file may be used to describe the format of an SCA structure parameter. Structure description files are very similar to Tuxedo viewfiles, with additional capabilities added for SCA.

Note: The use of structure description files is optional, and is needed only when FML field names corresponding to structure elements are different from the names of the structure elements, or when some other non-default structure related feature is required. If an application wants to make use of an Associated Length Member, an Associated Count Member, or an application-specified default value for a structure element, it may choose to make use of a structure description file.

If no structure description file is provided for a particular structure, then the structure definition used in application code is used, and FML field names in SCA-ATMI mode are the same as structure element names. Since field numbers are generated automatically for SCA-SCA applications, these applications do not need to specify a structure description file.

The structure description file format is identical to the Tuxedo viewfile format, with the following exceptions:

- The type parameter in column 1 allows the additional values `bool`, `unsignedchar`, `signedchar`, `wchar_t`, `unsignedint`, `unsignedlong`, `longlong`, `unsignedlonglong`, `longdouble`, and `struct`.
- If the value in column1 is `struct`, then the `cname` value in column 2 is the name of a previously defined VIEW that describes a nested structure. In this case, the count value in column 4 may optionally be specified to specify the number of occurrences of the nested structure.

If a structure described in a structure description file is converted to (or from) an FML32 or FML buffer at runtime in an SCA-ATMI application, then the name of the corresponding FML field is the `fbyname` value specified in column 3, if any, and is the `cname` value specified in column 2 (if no value is specified in column 3). When compiled, the structure description file produces a binary structure description file as shown in [Listing 6-23](#). The binary structure header file is shown in shown in [Listing 6-24](#).

Note: In an SCA-SCA application, FML32 field numbers are generated automatically.

Listing 6-23 SCA Structure Description File

```

VIEW empname
#TYPE          CNAME          FBNAME          COUNT          FLAG          SIZE          NULL
string         fname          EMP_FNAME       1              -             25            -
char           minit          EMP_MINI        1              -             -             -
string         lname          EMP_LNAME       1              -             25            -
END

VIEW emp
struct          empname          ename          1              -             -             -
unsignedlong    id              EMP_ID         1              -             -             -
long           ssn              EMP_SSN        1              -             -             -
double         salaryhist    EMP_SAL        10            -             -             -
END

```

Listing 6-24 Binary Structure Header File

```

struct empname {
    char fname[25];
    char minit;
    char lname[25];
};

struct emp {
    struct empname ename;
    unsigned long id;
    long ssn;
    double salaryhist[10];
}

```

The `scastructc32` and `scastructc` commands are used to convert a source structure description file into a binary structure description file and to generate a header file describing the structure(s) in the structure description file. The `scastructdis32` and `scastructdis`

commands accept the same arguments as `viewdis32` and `viewdis`. For more information, see the [Oracle SALT Command Reference](#).

Notes: `scastructc32` and `scastructc` generate a binary file with suffix `.V` on Unix and suffix `.VV` on Windows.

If the structure description file contains no SCA extensions that are not available in Tuxedo views, then the magic value for the binary structure description file shall be the same as the magic value used by `viewc32`. If any SCA specific extensions are used, then a different magic value shall be used for the binary structure description file.

Using `tuxscagen` to Generate Structures

When invoked with the option `-S`, `tuxscagen` generates a structure for any function parameter or return value that would otherwise have been passed using `DataObjectPtr`.

Note: If `tuxscagen -S` is run, then simple data types are generated just as they would have been if `tuxscagen` were run without the `-S` option. It is possible to mix simple data types, structures, and other complex data types within a single metadata repository. In order to use simple data types in an application that also uses structures, it is not necessary to run `tuxscagen` without `-S`.

SCA Remote Protocol Support

SCA Tuxedo invocation supports the following remote protocols:

- [/WS](#)
- [/Domains](#)

[/WS](#)

SCA invocations made using the SCA container have the capability of being performed using the Tuxedo WorkStation protocol (`/WS`). This is accomplished by specifying the value `WorkStation` (not abbreviated so as not to confuse it with `WebServices`) in the `<remoteAccess>` element of the `<binding.atmi>` element.

Only reference-type invocations are available in this mode. Service-type invocations may be performed using the `/WS` transparently (there is no difference in behavior or configuration, and setting the `<remoteAccess>` element to `WorkStation` for an SCA service has no effect).

Since native and `WorkStation` libraries cannot be mixed within the same process, client processes must be built differently depending on the type of remote access chosen.

Note: When using the value `propagatesTransaction` in `/binding.atmi/@requires`, the behavior of the ATMI binding does not actually perform any transaction propagation. It actually starts a transaction, since the use of this protocol is reserved for client-side access to Tuxedo (SCA or non-SCA) applications only. For more information, see [ATMI Binding](#).

/Domains

SCA invocations made using the SCA container have the capability of being performed using the Tuxedo /Domains protocol. No additional configurations are necessary on `<binding.atmi>` declarations in SCDL files.

Note: /Domains interoperability configuration is controlled by the Tuxedo administrator.

The SCA service name configured for Tuxedo /Domains is as follows:

- SCA -> SCA mode - `/binding.atmi/service/@name` attribute followed by a '/' and method name
- Legacy mode (SCA -> Tux interop mode) - `/binding.atmi/service/@name` attribute.

For more information, see [SCA and Tuxedo Interoperability](#).

SCA Binding

Oracle SALT supports

- [ATMI Binding](#)
- [Java ATMI \(JATMI\) Binding](#)
- [Python and Ruby Binding](#)
- [Web Services Binding](#)

ATMI Binding

Tuxedo communications are configured in SCDL using a `<binding.atmi>` element. This allows you to specify configuration elements specific to the ATMI transport, such as the location of the TUXCONFIG file, the native Tuxedo buffer types used, Tuxedo-specific authentication or /WS (WorkStation) configuration elements, etc.

[Listing 6-25](#) shows a summary of the `<binding.atmi>` element.

Note: ? refers to a parameter that can be specified 0 or 1 times.

* refers to a parameter that can be specified 0 or more times.

For more information, see [Appendix F: Oracle SALT SCA ATMI Binding Reference](#) in the *Oracle SALT Reference Guide*.

Listing 6-25 ATMI Binding Pseudoschema

```
<binding.atmi requires="transactionalintent legacyintent"?>
  <tuxconfig>...</tuxconfig>?
  <map target="name">...</map>*
  <serviceType target="name">...</serviceType>*
  <inputBufferType target="name">...</inputBufferType>*
  <outputBufferType target="name">...</outputBufferType>*
  <errorBufferType target="name">...</errorBufferType>*
  <workStationParameters>?
    <networkAddress>...</networkAddress>?
    <secPrincipalName>...</secPrincipalName>?
    <secPrincipalLocation>...</secPrincipalLocation>?
    <secPrincipalPassId>...</secPrincipalPassId>?
    <encryptBits>...</encryptBits>?
  </workStationParameters>
  <authentication>?
    <userName>...</userName>?
    <clientName>...</clientName>?
    <groupName>...</groupName>?
    <passwordIdentifier>...</passwordIdentifier>?
    <userPasswordIdentifier>...
                                </userPasswordIdentifier>?
  </authentication>
  <fieldTablesLocation>...</fieldTablesLocation>?
  <fieldTables>...</fieldTables>?
  <fieldTablesLocation32>...</fieldTablesLocation32>?
  <fieldTables32>...</fieldTables32>?
  <viewFilesLocation>...</viewFilesLocation>?
  <viewFiles>...</viewFiles>?
  <viewFilesLocation32>...</viewFilesLocation32>?
```

```
<viewFiles32>...</viewFiles32>?
<remoteAccess>...</remoteAccess>?
<transaction timeout="xsd:long"/>?
</binding.atmi>
```

Java ATMI (JATMI) Binding

Java ATMI (JATMI) binding allows SCA clients written in Java to call Tuxedo services or SCA components. It provides one-way invocation of Tuxedo services based on the Tuxedo WorkStation protocol (/WS). The invocation is for outbound communication only from a Java environment to Tuxedo application acting as a server. Apart from a composite file for SCDL binding declarations, no external configuration is necessary. The service name, workstation address and authentication data are provided in the binding declaration.

Note: SSL is supported through the [Oracle 11gR1 JCA Adapter](#). LLE is not currently supported.

Most of the Tuxedo CPP ATMI binding elements support JATMI binding and have the same usage. However, due to different underlying technology and running environment differences, some elements are not supported and some that are supported but have different element names.

The following Tuxedo CPP ATMI binding elements are not supported:

- `binding.atmi/tuxconfig`
- `binding.atmi/fieldTablesLocation`
- `binding.atmi/fieldTablesLocation32`
- `binding.atmi/viewFilesLocation`
- `binding.atmi/viewFilesLocation32`
- `binding.atmi/transaction`

The following Tuxedo CPP ATMI binding `workStationParameters` elements are not supported:

- `binding.atmi/workStationParameters/secPrincipalName`
- `binding.atmi/workStationParameters/secPrincipalLocation`
- `binding.atmi/workStationParameters/secPrincipalPassId`
- `binding.atmi/workStationParameters/encryptBits`

The following Tuxedo CPP ATMI binding element is supported in a limited fashion.

- `binding.atmi/remoteAccess`

Note: Only the value "WorkStation" is allowed. If not specified, "WorkStation" is assumed.

All the classes in the elements mentioned below must be specified in Java CLASSPATH:

- `binding.atmi/fieldTables` - Specifies a comma-separated list of Java classes that are extended from the `weblogic.wtc.jatmi.TypedFML` base class.
- `binding.atmi/fieldTables32` - Specifies a comma-separated list of Java classes that are extended from the `weblogic.wtc.jatmi.TypedFML32` base class.
- `binding.atmi/viewFiles` - Specifies a comma-separated list of Java classes that are extended from the `weblogic.wtc.jatmi.TypedView` base class. These derived classes usually are generated from a Tuxedo VIEW file using the `weblogic.wtc.jatmi.viewj` compiler. These also includes derived from `weblogic.wtc.jatmi.TypedXCType` and `weblogic.wtc.jatmi.TypedXCommon`.

For more information, see [How to Use the viewj Compiler](#) in the Tuxedo WebLogic Tuxedo Connector Programmer's Guide.

- `binding.atmi/viewFiles32` - Specifies a comma-separated list of Java classes that are extended from the `weblogic.wtc.jatmi.TypedView32` base class. These derived classes usually are also generated from a Tuxedo VIEW file using the `weblogic.wtc.jatmi.viewj32` compiler.

[Listing 6-26](#) shows an example of composite file for binding declaration of a Tuxedo service named "ECHO".

Listing 6-26 ECHO Composite File

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.oxa.org/xmlns/sca/1.0"
xmlns:f="binding-atmi.xsd"
name="ECHO">
  <reference name="ECHO" promote="EchoComponent/ECHO">
    <interface.java interface="com.abc.sca.jclient.Echo" />
    <f:binding.atmi requires="legacy">
      <f:inputBufferType target="echoStr">STRING</f:inputBufferType>
      <f:outputBufferType target="echoStr">STRING</f:outputBufferType>
      <f:errorBufferType target="echoStr">STRING</f:errorBufferType>
    </f:binding.atmi>
  </reference>
</composite>
```

```
        <f:workStationParameters>

<f:networkAddress>//STRIATUM:9999, //STRIATUM:1881</f:networkAddr
ess>
        </f:workStationParameters>
        <f:remoteAccess>WorkStation</f:remoteAccess>
    </f:binding.atmi>
</reference>
<component name="EchoComponent">
    <implementation.java class="com.abc.sca.jclient.EchoComponentImpl"
/>
    </component>
</component>
```

[Listing 6-27](#) shows the interface for the example mentioned in [Listing 6-26](#).

Listing 6-27 ECHO Interface

```
package com.abc.sca.jclient;

import com.oracle.jatmi.AtmiBindingException;

public interface Echo {
    String echoStr(String requestString) throws AtmiBindingException;
}
```

[Listing 6-28](#) shows an example of an SCA client implementation.

Listing 6-28 SCA Client Implementation

```
package com.abc.sca.jclient;

import org.osoa.sca.annotations.Constructor;
```

```

import org.osoa.sca.annotations.Reference;
import com.oracle.jatmi.AtmiBindingException;

/**
 * A simple client component that uses a reference with a JATMI binding.
 */
public class EchoComponentImpl implements Echo {

    private Echo echoReference;

    @Constructor
    public EchoComponentImpl(@Reference(name = "ECHO", required = true)
Echo
    echoReference) {
        this.echoReference = echoReference;
    }

    public String echoStr(String requestString) throws
AtmiBindingException {
        return echoReference.echoStr(requestString);
    }
}

```

Python and Ruby Binding

The Python and Ruby language extensions are binding-independent, meaning that binding extensions are not aware of the language of clients or components. Language extensions are not aware of the binding used.

Binding extensions are not modified to comply with Python and Ruby program support. Note the following:

- If the data types defined in Python or Ruby binding do not match the Python or Ruby source files, then Oracle SALT will throw an exception.
- If a binding is configured with a data type that does not match what the Python or Ruby component is designed to handle, an exception is thrown by the Python or Ruby runtime

(for example, `binding.atmi` configured with STRING Tuxedo buffers and a Python function handling numerical data).

- For a Python or Ruby client code mismatch with what binding is configured with, an exception occurs originating from the binding code.
- Since Python and Ruby code is not compiled, any configuration mismatch between binding and component/client can only be detected at runtime.
- Python or Ruby programs with a *composite* scope require a Tuxedo server reload when the script is modified. A *stateless* scope allows dynamic reloading of modified scripts.
- In order to expose Python or Ruby scripts as Web services, the `SCAHOST` command must use the `-w` option in order to load the correct service binding during initialization.

Note: `SCAHOST` does not allow mixing both ATMI and Web services binding types in one `SCAHOST` instance.

For more information, see the [Oracle SALT Command Reference](#).

- `TMMETADATA` server is required in order to expose Python and Ruby components.

Python and Ruby Binding Limitations

Using Python and Ruby bindings have the following limitations:

- When using the ATMI binding for interoperability calls (that is, when `requires="legacy"` is set), mixing named and non-named parameters is not allowed (for example, Python: `def func(a, *b, **c)`, Ruby: `def func(a, *b, hash)`), since there is no mechanism to restore the parameter names.

The names of the parameters must be configured in FML32 tables (ATMI binding), or by way of WSDL (Web services binding). It is not possible to interoperate with lists of non-named parameters because such calls cannot be accurately mapped to C++ or WSDL interfaces due to the lack of guaranteed ordering of FML/FML32 Tuxedo buffers.

The supported modes are:

- Multiple parameters: `def func(a, b, c)` (same syntax for Python and Ruby)
- Multiple parameters and list of parameters: `def func(a, *b)` (same syntax for Python and Ruby)
- Dictionary or hash: Python: `def func(**a)`, Ruby: `def func(hash)`

Note: Python parameters defined with `**` are considered named parameters. Ruby parameters defined with `hash` are considered named parameters. For more information, see [Python Parameters](#) and [Ruby Parameters](#).

- In SCA to SCA mode, the above limitation still concerns named parameters since the order of elements in a Python dictionary or Ruby hash is not guaranteed. To transmit a Python dictionary or Ruby hash, you must work in "legacy" mode.
- In SCA to SCA mode, using lists of parameters (excluding dictionaries or hashes) are supported since Tuxedo Service Metadata interfaces describe service-side lists of parameters/types (on the reference side parameters/types are self-described at runtime).
- Unicode strings are not supported; accordingly MBSTRING buffers or FLD_MBSTRING fields are not supported.

Web Services Binding

The Web services binding (`binding.ws`) leverages previously existing Oracle SALT capabilities by funneling Web service traffic through the GWWS gateway. SCA components are hosted in Tuxedo servers, and communications to and from those servers are performed using the GWWS gateway.

SCA clients using a Web services binding remain unchanged whether the server is running in a Tuxedo environment or a native Tuscany environment (for example, exposing the component using the Axis2 Web services binding).

Note: HTTPS is not currently supported.

When SCA components are exposed using the Web services binding (`binding.ws`), tooling performs the generation of WSDF information, metadata entries and FML32 field definitions.

When SCDL code of SCA components to be hosted in a Tuxedo domain (for example, service elements) contains `<binding.ws>` elements, the `buildscaserver` command generates an WSDF entry in a file named `service.wsdf` where 'service' is the name of the service exposed. An accompanying `service.mif` and `service.fml32` field table files are also generated, based on the contents of the WSDL interface associated with the Web service. You must compose a WSDL interface. If no WSDL interface is found, an error message is generated.

Web services accessed from a Tuxedo domain using a Web services binding (for example, reference elements found in SCDL) require the following manual configuration steps:

1. Convert the WSDL file into a WSDF entry by using the `wsdlcvt` tool. Simultaneously, a Service Metadata Entry file (`.mif`), and `fml32` mapping file are generated.

2. Make sure that the UBB source has the TMMETADATA and GWWS servers configured
3. Import the WSDL file into the SALTDEPLOY file
4. Convert the SALTDEPLOY file into binary using `wsloadcf`.
5. Load the Service Metadata Entry file (`.mif`) into the Service Metadata Repository using the `tmloadrepos` command.
6. Boot (or re-boot) the GWWS process to initiate the new deployment.

The Web services binding reference extension initiates the Web services call.

[Listing 6-29](#) shows an SCA component service exposed as a Web service.

Listing 6-29 Example SCA Component Service Exposed as a Web Service

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  name="bigbank.account">
  ...
  <service name="AccountService">
    <interface.wSDL interface="http://www.bigbank.com/AccountService
      #wSDL.interface(AccountService)"/>
    <binding.ws/>
    <reference>AccountServiceComponent</reference>
  </service>

  <component name="AccountServiceComponent">
    <implementation.cpp
      library="Account" header="AccountServiceImpl.h"/>
    <reference name="accountDataService">
      AccountDataServiceComponent
    </reference>
  </component>
  ...
</composite>
```

The steps required to expose the corresponding service are as follows:

1. Compose a WSDL interface matching the component interface.
2. Use `buildscacomponent` to build the application component runtime, similar to building a regular SCA component.
3. `buildscaserver -w` is used to convert SCDL code into a WSDF entry, and produce a deployable server (Tuxedo server + library + SCDL).

The service from the above SCDL creates a WSDF entry as shown in [Listing 6-30](#).

Listing 6-30 WSDF Entry

```
<Definition>
  <WSBinding id="AccountService_binding">
    <ServiceGroup id="AccountService">
      <Service name="TuxAccountService" />
    </ServiceGroup>
  </WSBinding>
</Definition>
```

4. `buildscaserver -w` also constructs a Service Metadata Repository entry based by parsing the SCDL and interface. The interface needs to be in WSDL form, and manually-composed in this release.
5. Make sure that the UBB source has the TMMETADATA and GWWS servers configured.
6. The Service Metadata Repository entry is loaded into the Service Metadata Repository using the `tmloadrepos` command.
7. The WSDF file must be imported into the SALTDEPLOY file and SALTDEPLOY converted into binary using `wsloadcf`.
8. The Service Metadata Entry file (`.mif`) is loaded into the Service Metadata Repository.
9. The Tuxedo server hosting the Web service is booted and made available.
10. The GWWS is rebooted to take into account the new deployment.

These steps are required, in addition to the SALTDEPLOY configuration, in order to set up the GWWS gateway for Web services processing (for example, configuration of `GWInstance`,

Server Level Properties, etc.). When completed, Web service clients (SCA or other) have access to the Web service.

[Listing 6-31](#) shows a reference accessing a Web service.

Listing 6-31 Example Reference Accessing a Web Service

```
<composite xmlns="http://www.oxa.org/xmlns/sca/1.0"
  name="bigbank.account">
  ...
  <reference name="StockQuoteWebService">
    <interface.wSDL interface="http://www.webserviceX.NET/#
      wSDL.interface (StockQuoteSoap) "/>
    <binding.ws endpoint="http://www.webserviceX.NET/#
      wSDL.endpoint (StockQuote/StockQuoteSoap) "/>
  </reference>
  ...
</composite>
```

The steps required to access the Web service are as follows:

1. A WSDL file is necessary. This is usually published by the Web Service provider.
2. The WSDL file must be converted into a WSDF entry using the `wSDLcvT` tool. At the same time a Service Metadata Entry file (`.mif`), and `fml32` mapping file is generated.
3. The WSDF file must be imported into the SALTDEPLOY file and SALTDEPLOY converted into binary using `wsloadcf`.
4. The Service Metadata Entry file (`.mif`) is loaded into the Service Metadata Repository using the `tmloadrepos` command.
5. The GWWS process is rebooted to take into account the new deployment.

These steps are required, in addition to the SALTDEPLOY configuration, in order to set up the GWWS gateway for Web services processing (for example, configuration of `GWInstance`, Server Level Properties, etc.). When completed, the SCA client has access to the Web service.

The process is the same, whether the client is stand-alone SCA program or an SCA component (already a server) referencing another SCA component via the Web service binding.

SCA Data Type Mapping

Using ATMI binding leverages the Tuxedo infrastructure. Data exchanged between SCA components, or Tuxedo clients/services and SCA clients/components is performed using Tuxedo typed buffers. [Table 6-1](#) through [Table 6-10](#) summarize the correspondence between native types and Tuxedo buffers/types, as well as SOAP types when applicable.

In the example shown in [Listing 6-32](#), implementations send and receive a Tuxedo STRING buffer. To the software (binding and reference extension implementations), the determination of the actual Tuxedo buffer to be used is provided by the contents of the `/binding.atmi/inputBufferType`, `/binding.atmi/outputBufferType`, or `/binding.atmi/errorBufferType` elements in the SCDL configuration, and the type of buffer returned (or sent) by a server (or client). It does not matter whether client or server is an ATMI program or an SCA component.

Notice that the Tuxedo `simpapp` service has its own namespace within `namespace services`. A C++ method `toupper` is associated with this service.

Listing 6-32 C++ Interface Example

```
#include <string>
namespace services
{
    namespace simpapp
    {
        /**
         * business interface
         */
        class ToupperService
        {
        public:

            virtual std::string
                toupper(const std::string inputString) = 0;
        };

    } // End simpapp
} // End services
```

The following data type mapping rules apply:

- [Run-Time Data Type Mapping](#)
- [SCA Utility Data Type Mapping](#)

Run-Time Data Type Mapping

- [Simple Tuxedo Buffer Data Mapping](#)
- [Complex Return Type Mapping](#)
- [Complex Tuxedo Buffer Data Mapping](#)

Simple Tuxedo Buffer Data Mapping

The following are considered to be simple Tuxedo buffers:

- STRING
- CARRAY (and X_OCTET)
- MBSTRING
- XML

[Table 6-1](#) lists simple Tuxedo buffer types that are mapped to SCA binding.

Table 6-1 Simple Tuxedo Buffer Type Data Mapping

C++ or STL Type	Java Type	Tuxedo Buffer Type	Notes
char*, char array or std::string	java.lang.String	STRING	
CARRAY_T	byte[] or java.lang.Byte[]	CARRAY	
X_OCTET_T	byte[] or java.lang.Byte[]	X_OCTET	

Table 6-1 Simple Tuxedo Buffer Type Data Mapping

C++ or STL Type	Java Type	Tuxedo Buffer Type	Notes
XML_T	byte[] or java.lang.Byte[]	XML	This type is passed as a C++ array within the data element of struct XML or as an array of java bytes. It is transformed to SDO.
wchar_t * or wchar_t array	N/A	MBSTRING	See Multibyte String Data Mapping
std::wstring	java.lang.String	MBSTRING	See Multibyte String Data Mapping

When a service called by an SCA client returns successfully, a pointer to the service return data is passed back to the Proxy stub generated by `buildscaclient`. The Proxy stub then de-references this pointer and returns the data to the application.

Table 6-1 can be interpreted as follows:

- When the reference or service binding extension runtime sees a Tuxedo STRING buffer, it looks for either a `char*`, `char` array, `std::string` parameter or `return` type (depending on the direction). If a different type is found, an exception is thrown with a message explaining what happened.
- When the reference or service binding extension runtime sees a `char*` (for example) as a single parameter or return type, it looks for STRING as the buffer type in the `binding.atmi` element. If a different Tuxedo buffer type is found, an exception is thrown with a message explaining what happened.

Multibyte String Data Mapping

Tuxedo uses multibyte strings to represent multibyte character data with encoding names based on `iconv` (as defined by Tuxedo). C++ uses a `wstring`, `wchar_t*`, or `wchar_t[]` data type to represent multibyte character data with encoding names (as defined by the C++ library).

Tuxedo and C++ sometimes use different names to represent a particular multibyte encoding. Mapping between Tuxedo encoding names and C++ encoding names is as follows:

- Receiving a Multibyte String Buffer

When an SCA client or server receives an MBSTRING buffer or an FML32 buffer with a FLD_MBSTRING field, it considers the encoding for that multibyte string to be the first locale from the following cases:

- a. Locale associated with the FLD_MBSTRING field, if present.

Note: For more information, see [Table 6-2](#).

- b. Locale associated with the MBSTRING or FML32 buffer.
- c. Locale set in the environment of the SCA client or server.

If case a or b is matched, Tuxedo invokes the `setlocale()` function for locale type `LC_CTYPE` with the locale for the received buffer. If `setlocale()` fails (indicating there is no such locale) and an alternate name has been associated with this locale in the optional `$TUXDIR/locale/setlocale_alias` file, Tuxedo attempts to set the `LC_CTYPE` locale to the alternate locale.

The `$TUXDIR/locale/setlocale_alias` file may be optionally created by the Tuxedo administrator. If present, it contains a mapping of Tuxedo MBSTRING codeset names to an equivalent operating system locale accepted by the `setlocale()` function.

Lines consist of a Tuxedo MBSTRING codeset name followed by whitespace and an OS locale name. Only the first line in the file corresponding to a particular MBSTRING codeset name are considered. Comment lines begin with `#`.

The `$TUXDIR/locale/setlocale_alias` file is used by the SALT SCA software when converting MBSTRING data into C++ `wstring` or `wchar_t[]` data. If `setlocale()` fails when using the Tuxedo MBSTRING codeset name, then the SALT SCA software attempts to use the alias name, if present. For example, if the file contains a line `'GB2312 zh_CN.GB2312'` then if `setlocale(LC_CTYPE, 'GB2312')` fails, the SALT SCA software attempts `setlocale(LC_CTYPE, 'zh_CN.GB2312')`.

- **Sending a Multibyte String Buffer**

When an SCA client or server converts a `wstring`, `wchar_t[]`, or `wchar_t*` to an MBSTRING buffer or a FLD_MBSTRING field, it uses the `TPMBENC` environment variable value as the locale to set when converting from C++ wide characters to a multibyte string. If the operating system does not recognize this locale, Tuxedo uses the alternate locale from the `$TUXDIR/locale/setlocale_alias` file, if any.

Note: It is possible to transmit multibyte data retrieved from an MBSTRING buffer, an FML32 FLD_MBSTRING field, or a VIEW32 mbstring field. It is also possible to transmit multibyte data entered using the SDO `setString()` method.

However, it is not possible to enter multibyte characters directly into an XML document and transmit this data via SALT. This is because multibyte characters entered in XML documents are transcoded into multibyte strings, and SDO uses `wchar_t` arrays to represent multibyte characters.

Complex Return Type Mapping

The following C++ built-in types (used as return types) are considered complex and automatically encapsulated in an FML/FML32 buffer as a single generic field following the complex buffer mapping rules described in [Complex Tuxedo Buffer Data Mapping](#). This mechanism addresses the need for returning types where a corresponding Tuxedo buffer cannot be used.

Note: Interfaces returning any of the built-in types assume that FML/FML32 is the output buffer type. The name of this generic field is `TUX_RTNdatype` based on the type of data being returned. `TUX_RTNdatype` fields are defined in the `Usysflds.h/Usysfld32.h` and `Usysflds/Usysfld32` shipped with Oracle Tuxedo.

- `bool` : maps to `TUX_RTNCHAR` field
- `char`: maps to `TUX_RTNCHAR` field
- `signed char`: maps to `TUX_RTNCHAR` field
- `unsigned char`: maps to `TUX_RTNCHAR` field
- `short`: maps to `TUX_RTNSHORT` field
- `unsigned short`: maps to `TUX_RTNSHORT` field
- `int`: maps to `TUX_RTNLONG` field
- `unsigned int`: maps to `TUX_RTNLONG` field
- `long`: maps to `TUX_RTNLONG` field
- `unsigned long`: maps to `TUX_RTNLONG` field
- `long long`: (maps to `TUX_RTNLONG` field
- `unsigned long long`: maps to `TUX_RTNLONG` field
- `float`: maps to `TUX_RTNFLOAT` field
- `double`: maps to `TUX_RTNDOUBLE` field

- long double: maps to TUX_RTNDOUBLE field

Complex Tuxedo Buffer Data Mapping

The following are considered to be complex Tuxedo buffers:

- FML
- FML32
- VIEW (and X_* equivalents)
- VIEW32

[Table 6-2](#) lists the complex Tuxedo buffer types that are mapped to SCA binding.

For FML and FML32 buffers, parameter names in interfaces must correspond to field names, and follow the restrictions that apply to Tuxedo fields (length, characters allowed). When these interfaces are generated from metadata using `tuxscagen(1)`, the generated code contains the properly formatted parameter names.

If an application manually develops interfaces without parameter names, manually develops interfaces that are otherwise incorrect, or makes incompatible changes to SALT generated interfaces, then incorrect results are likely to occur.

VIEW (and X_* equivalents) and VIEW32 buffers require the use of SDO `DataObject` wrappers.

[Listing 6-33](#) shows an interface example. The associated field definitions (following the interface) must be present in the process environment.

Table 6-2 Complex Tuxedo Buffer Type Data Mapping

C++, STL, or SDO type	Java Type	Tuxedo field type	Tuxedo view type	Notes
bool	boolean or java.lang.Boolean	FLD_CHAR	char	Maps to 'T' or 'F'. (This matches the mapping used elsewhere in SALT.)
char, signed char, or unsigned char	byte or java.lang.Byte	FLD_CHAR	char	

Table 6-2 Complex Tuxedo Buffer Type Data Mapping

C++, STL, or SDO type	Java Type	Tuxedo field type	Tuxedo view type	Notes
short or unsigned short	short or java.lang.Short	FLD_SHORT	short	An unsigned short is cast to a short before being converted to FLD_SHORT or short.
int or unsigned int	int or java.lang.Integer	FLD_LONG	int	An unsigned int being converted to FML or FML32 is cast to a long before being converted to FLD_LONG or long. An unsigned int being converted to a VIEW or VIEW32 member is cast to an int.
long or unsigned long	long or java.lang.Long	FLD_LONG	long	An exception is thrown if the value of a 64-bit long does not fit into a FLD_LONG or long on a 32-bit platform. An unsigned long is cast to long before being converted to FLD_LONG or long.
long long or unsigned long long	N/A	FLD_LONG	long	An exception is thrown if the data value does not fit within a FLD_LONG or long. An unsigned long long is cast to long long before being converted to FLD_LONG or long.
float	float or java.lang.Float	FLD_FLOAT	float	

Table 6-2 Complex Tuxedo Buffer Type Data Mapping

C++, STL, or SDO type	Java Type	Tuxedo field type	Tuxedo view type	Notes
double	double or java.lang.Double	FLD_DOUBLE	double	
long double	N/A	FLD_DOUBLE	double	
char* or char array	N/A	FLD_STRING	string	
std::string	java.lang.String	FLD_STRING	string	
CARRAY_T or X_OCTET_T	class CARRAY	FLD_CARRAY	carray	Will map externally following GWWS rules. This departs from the OSOA spec. (which does not support them), and should be considered an improvement.
Bytes	N/A	FLD_CARRAY	Carray	This mapping is used when part of a DataObject
wchar_t* or wchar_t array	N/A	FLD_MBSTRING (FML32 only)	mbstring (VIEW32 only)	(Java char is Unicode and can range from -32768 to +32767.) See also Multibyte String Data Mapping
std::wstring	java.lang.String	FLD_MBSTRING (FML32 only)	mbstring (VIEW32 only)	See also Multibyte String Data Mapping

Table 6-2 Complex Tuxedo Buffer Type Data Mapping

C++, STL, or SDO type	Java Type	Tuxedo field type	Tuxedo view type	Notes
<code>commonj::sdo::DataObjectPtr</code>	<code>TypedFML32</code>	<code>FLD_FML32</code> (FML32 only)	N/A	Generate a data transformation exception, which is translated to an <code>ATMIBindingException</code> before being returned to the application, when: <ul style="list-style-type: none"> • Attempting to add such a field in a Tuxedo buffer other than FML32 • The data object is not typed (i.e., there is no corresponding schema describing it). See also Multibyte String Data Mapping
<code>commonj::sdo::DataObjectPtr</code>	<code>TypedView32</code>	<code>FLD_VIEW32</code> (FML32 only)	N/A	See also Multibyte String Data Mapping
<code>struct structurename</code>	N/A	<code>FLD_FML32</code> (FML32 only)	<code>structurename</code> <code>e</code>	See also SCA Structure Support

Listing 6-33 Interface Example

```

...
int myService(int param1, float param2); ...
Field table definitions
#name          number type          flag comment
#-----
param1 20      int           -      Parameter 1
param2 30      float         -      Parameter 2
...

```

SDO Mapping

C++ method prototypes that use `commonj::sdo::DataObjectPtr` objects as parameter or return types are mapped to an FML, FML32, VIEW, or VIEW32 buffer.

You must provide an XML schema that describes the SDO object. The schema is made available to the service or reference extension runtime by placing the schema file (`.xsd` file) in the same location as the SCDL composite file that contains the reference or service definition affected. The schema is used internally to associate element names and field names.

Note: When using `view` or `view32`, a schema type (for example, `complexType`) which name matches the `view` or `view32` used is required.

For more information, see `mkfldfromschema` and `mkfld32fromschema` in the *SALT 11g Release 1 (11.1.1.1.0) Command Reference*.

For example, a C++ method prototype defined in a header such as:

```
long myMethod(commonj::sdo::DataObjectPtr data);
```

[Listing 6-34](#) shows the associated schema.

Listing 6-34 Schema

```
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
  xmlns="http://www.example.com/myExample"
  targetNamespace="http://www.example.com/myExample">

  <xsd:element name="bike" type="BikeType"/>
  <xsd:element name="comment" type="xsd:string"/>

  <xsd:complexType name="BikeType">
    <xsd:sequence>
      <xsd:element name="serialNO" type="xsd:string"/>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="type" type="xsd:string"/>
      <xsd:element name="price" type="xsd:float"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Table 6-3 shows the generated field table.

Table 6-3 Generated Field Tables

NAME	NUMBER	TYPE	FLAG	Comment
bike	20	fml32	-	
comment	30	string	-	
serialNO	40	string	-	
name	50	string	-	
type	60	string	-	
price	70	float	-	

The following restrictions in XML schemas apply:

- Attributes cannot be specified and are ignored if specified
- Values in restrictions are ignored (their meaning is application-related), only the field name and type are generated
- When using XML schema types, only signed integral types are supported. See "SDO C++ Specification" for a list of available SDO primitive types.

SCA Utility Data Type Mapping

The `scatuxgen` and `tuxscagen` utilities are used to generate manual SCA data type mapping. The `scatuxgen` mapping rules are as follows:

- [C++ Parameter/Return Type and Tuxedo Buffer Type Mapping](#)
- [C++ Parameter Type and Tuxedo Parameter Type Mapping](#)
- [C++ Parameter Type and Tuxedo Complex Type Mapping](#)
- [Parameter and Return Types to Parameter-Level Keyword Restrictions](#)

Note: The mapping rules for `tuxscagen` are executed in the reverse direction (Tuxedo Buffer Type -> C++ Parameter Type).

C++ Parameter/Return Type and Tuxedo Buffer Type Mapping

Table 6-4 shows the correspondence between parameter/return types and Tuxedo buffer types (inbuf service-level keyword).

Table 6-4 'inbuf' Keyword Buffer Type Mapping Table

C++ Parameter Type	TUXEDO Buffer Type
std::string or char*	STRING
struct carray_t	CARRAY
char	FML32
short	FML32
int	FML32
long	FML32
float	FML32
double	FML32
wchar_t[]	MBSTRING
struct xml_t	XML
struct x_octet_t	X_OCTET
commonj::sdo::DataObjectPtr	X_COMMON, X_C_TYPE, VIEW, VIEW32, FML, or FML32 depending on inputBufferType setting
struct structurename	X_COMMON, X_C_TYPE, VIEW, VIEW32, FML, or FML32 depending on inputBufferType setting
multiple parameters, or one commonj::sdo::DataObjectPtr or struct structurename and no binding.atmi or no corresponding inputBufferType and the input buffer is not specified using a command line option	FML32

Table 6-5 shows the correspondence between parameter/return types and Tuxedo buffer types (outbuf or err buf service-level keywords).

Table 6-5 outbuf' or 'errbuf' Keyword Buffer Type Mapping Table

C++ Return Type	TUXEDO Buffer Type
<code>std::string</code> or <code>char*</code>	STRING
<code>struct carray_t</code>	CARRAY
<code>char</code>	FML32
<code>short</code>	FML32
<code>int</code>	FML32
<code>long</code>	FML32
<code>float</code>	FML33
<code>double</code>	FML32
<code>wchar_t[], wstring</code>	MBSTRING
<code>struct xml_t</code>	XML
<code>struct x_octet_t</code>	X_OCTET
<code>commonj::sdo::DataObjectPtr</code>	X_COMMON, X_C_TYPE, VIEW, VIEW32, FML or FML32 depending on the <code>binding.atmi/outputBufferType</code> or <code>binding.atmi/errorBufferType</code> setting.
<code>commonj::sdo::DataObjectPtr</code>	FML32 if no <code>binding.atmi</code> is set, or <code>binding.atmi</code> is set and <code>binding.atmi/outputBufferType</code> or <code>binding.atmi/errorBufferType</code> aren't specified.
<code>struct structurename</code>	X_COMMON, X_C_TYPE, VIEW, VIEW32, FML or FML32 depending on the <code>binding.atmi/outputBufferType</code> or <code>binding.atmi/errorBufferType</code> setting.
<code>struct <i>structurename</i></code>	FML32 if no <code>binding.atmi</code> is set, or <code>binding.atmi</code> is set and <code>binding.atmi/outputBufferType</code> or <code>binding.atmi/errorBufferType</code> are not specified.

C++ Parameter Type and Tuxedo Parameter Type Mapping

Table 6-7 shows how `scatuxgen` handles interface parameter types and converts them to a Tuxedo Service Metadata Repository parameter-level keyword value when more than one parameter is used in the method signature.

Table 6-6 Parameter-Level/Field Type Mapping Table

C++ Parameter Data Type	Tuxedo Parameter-Level Keyword (FML FIELD Type)
<code>char</code>	<code>byte(FLD_CHAR)</code>
<code>short</code>	<code>short(FLD_SHORT)</code>
<code>int</code>	<code>integer(FLD_LONG)</code>
<code>long</code>	<code>integer(FLD_LONG)</code>
<code>float</code>	<code>float(FLD_FLOAT)</code>
<code>double</code>	<code>double(FLD_DOUBLE)</code>
<code>std::string</code> or <code>char *</code>	<code>string(FLD_STRING)</code>
<code>struct carray_t</code>	<code>carray(FLD_CARRAY)</code>
<code>std::wstring</code>	<code>mbstring(FLD_MBSTRING)</code>
<code>commonj::sdo::DataObjectPtr</code>	<code>fml32(FLD_FML32)</code>
<code>struct structurename</code>	<code>fml32(FLD_FML32)</code>

C++ Parameter Type and Tuxedo Complex Type Mapping

This section contains the following topics:

- [SDO Mapping](#)
- [C Struct Mapping](#)

SDO Mapping

When a method takes an SDO object as an argument, or returns an SDO object, for example as follows: `commonj::sdo::DataObjectPtr myMethod(commonj::sdo::DataObjectPtr input)`.

The corresponding runtime type may be described by an XML schema as shown in [Listing 6-35](#) and then referenced in the binding as shown in [Listing 6-36](#).

Listing 6-35 XML Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="tuxedo"
targetNamespace="tuxedo">

  <xsd:complexType name="BikeInventory">
    <xsd:sequence>
      <xsd:element name="BIKES" type="Bike"
        minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="STATUS" type="xsd:string" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Bike">
    <xsd:sequence>
      <xsd:element name="SERIALNO" type="xsd:string"/>
      <xsd:element name="SKU" type="xsd:string"/>
      <xsd:element name="NAME" type="xsd:string"/>
      <xsd:element name="TYPE" type="xsd:string"/>
      <xsd:element name="PRICE" type="xsd:float"/>
      <xsd:element name="SIZE" type="xsd:int"/>
      <xsd:element name="INSTOCK" type="xsd:string"/>
      <xsd:element name="ORDERDATE" type="xsd:string"/>
      <xsd:element name="COLOR" type="xsd:string"/>
      <xsd:element name="CURSERIALNO" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

Listing 6-36 Binding

```

...
<reference name="UBIK">
  <interface.cpp header="uBikeService.h" />
  <binding.atmi>
    <inputBufferType>FML32/Bike</inputBufferType>
    <outputBufferType>FML32/BikeInventory</outputBufferType>
  </binding.atmi>
</reference>
...

```

When such a schema is present, `scatuxgen` parses it and generates the corresponding parameter-level mapping entries as listed in [Table 6-7](#).

Table 6-7 Parameter-level/Field Type Mapping

XML Schema element type	Tuxedo Parameter-Level Keyword (FML FIELD Type)
xsd:byte	byte(FLD_CHAR)
xsd:short	short(FLD_SHORT)
xsd:int	integer(FLD_LONG)
xsd:long	integer(FLD_LONG)
xsd:float	float(FLD_FLOAT)
xsd:double	double(FLD_DOUBLE)
xsd:string	string(FLD_STRING)
xsd:string	mbstring(FLD_MBSTRING) when -t option is specified
xsd:base64binary	carray(FLD_CARRAY)
xsd:complexType	fml32(FLD_FML32)
xsd:minOccurs	requiredcount
xsd:maxOccurs	count

C Struct Mapping

When a method takes a C struct as an argument, or returns a C struct (for example, as shown in [Listing 6-37](#)), `scatuxgen` parses it and generates the corresponding parameter-level mapping entries listed in [Table 6-8](#).

Listing 6-37 C Struct

```

struct customer {
    char  firstname[80];
    char  lastname[80];
    char  address[240];
};

struct id {
    int  SSN;
    int  zipCode;
};

struct customer* myMethod(struct *id input);

```

Table 6-8 Parameter-Level/Field Type Mapping

Struct Member Type	Tuxedo Parameter-Level Keyword (FML FIELD Type)
char, unsigned char, signed char	byte(FLD_CHAR)
char []	string(FLD_STRING)
wchar_t []	mbstring(FLD_MBSTRING)
short, unsigned short	short(FLD_SHORT)
int, unsigned int	integer(FLD_LONG)

Table 6-8 Parameter-Level/Field Type Mapping

Struct Member Type	Tuxedo Parameter-Level Keyword (FML FIELD Type)
long, unsigned long, long long, unsigned long long	integer(FLD_LONG)
float	float(FLD_FLOAT)
double, long double	double(FLD_DOUBLE)
struct <i>nestedstructname</i> (for more information, see SCA Structure Support)	fml32 (FLD_FML32)
array type	count=requiredcount=array specifier

Parameter and Return Types to Parameter-Level Keyword Restrictions

For parameter-level keywords, the Tuxedo buffer type/parameter type restrictions are consistent with the contents expected by `tmloadrepos`. An error message is returned when an attempt to match any combinations that are not listed in [Table 6-9](#) and [Table 6-10](#).

Table 6-9 Tuxedo Buffer Type/Parameter Type Restrictions (Part 1)

Parameter Type / Tuxedo Buffer	byte(char)	short	integer	float	double	String
CARRAY						
FML	X	X	X	X	X	X
FML32	X	X	X	X	X	X
VIEW	X	X	X	X	X	X
VIEW32	X	X	X	X	X	X
X_COMMON		X		X		X
X_C_TYPE	X	X	X	X	X	X

Table 6-9 Tuxedo Buffer Type/Parameter Type Restrictions (Part 1)

Parameter Type / Tuxedo Buffer	byte(char)	short	integer	float	double	String
X_OCTET						
STRING						X
XML						X
MBSTRING						

Table 6-10 Tuxedo Buffer Type/Parameter Type Restrictions (Part 2)

Parameter Type / Tuxedo Buffer	carray	xml	view32	fml32	mbstring
CARRAY	X				
FML	X				
FML32	X	X	X	X	X
VIEW	X				
VIEW32	X				X
X_COMMON					
X_C_TYPE					
X_OCTET	X				
STRING					
XML		X			
MBSTRING	X				X

Python and Ruby Data Type Mapping

The following sections describe the supported data types in Python and Ruby clients or components with respect to the native, C/C++ based environment, and in order to give the

correspondence for writing the Oracle Tuxedo Service Metadata Repository interface required by the ATMI binding. Corresponding Oracle Tuxedo buffer and field type are also indicated for uses with the ATMI or Web Services binding.

- [Python Data Type Mapping](#)
- [Ruby Data Type Mapping](#)

Python Data Type Mapping

In Python, clients or components only use parameters and return values which types are listed in [Table 6-11](#). Multiple parameters are supported (in the same way that multiple parameters are supported in C++), using FML32 Tuxedo buffers.

Note: Arrays are not supported as they are not supported by bindings or the C++ language extension.

Table 6-11 Supported Python, C++ and Tuxedo Buffer Types

Python parameter(s) or Return Type	C/C++ Native Type	ATMI Binding Type Buffer type/Field Type
int	short, unsigned short	FML32/FLD_SHORT
long	short, unsigned short	FML32/FLD_SHORT
int	long, unsigned long	FML32/FLD_LONG
long	long, unsigned long	FML32/FLD_LONG
bool	bool	FML32/FLD_CHAR
float	float	FML32/FLD_FLOAT
float	double, long double	FML32/FLD_DOUBLE
string of length 1	char	FML32/FLD_CHAR
string	char *, std::string	STRING
xml	commonj::sdo::DataObject Ptr	FML32, VIEW, VIEW32

Notes: `int (short)`, `int (long)`, `float (float)` are allowed in the C++ to Python direction only. The Python runtime catches any overflow situation (e.g.: when copying a C++ long into a Python int).

In order to map a string of length 1 to a `char*/std::string/STRING`, the originating Python variable will have to have 2 ending zeroes, e.g.: `t = "a\x00"`.

Supported XML objects in Python must be `xml.etree.ElementTree` objects, (that is, the language extension converts `xml.etree.ElementTree` objects into `commonj::sdo::DataObjectPtr` objects, and `commonj::sdo::DataObjectPtr` objects into `xml.etree.ElementTree` objects).

Using lists and dictionaries are also supported, as detailed in [Python Parameters](#) and [Dictionaries](#).

Note: Lists and dictionaries are allowed as parameters, but are not allowed to be returned.

Some limitations concerning multiple parameters and lists will stand with respect to using bindings. For more information, see [Python and Ruby Binding](#).

Python Parameters

You can use the list notation (*) to pass an undetermined number of parameters to/from a Python program. For example:

```
def test(*params)
    for p in params:
        print "parameter:", p
```

and an example of call: `test(1, 2, 3, 4, 5)`

This notation is equivalent to having an actual list of parameters, such as:

```
def test(parm1, parm2, parm3, parm4, parm5)
    ...
```

Individual supported types are limited to the types listed in [Table 6-11](#).

Exposing a Python function as an SCA service with ATMI or Web services binding requires an interface. This interface is stored in the Tuxedo Service Metadata Repository as outlined in [Python and Ruby Component Programming](#).

When called, the Python function receives a list of parameters corresponding exactly to what the interface specifies. Any extra parameters passed by the client are ignored, and any type mismatch results in a data mapping exception.

Note: Using this notation is limited to local calls (no binding), or using ATMI binding between SCA components (that is, the `<binding.atmi>` element with no `requires="legacy"` attribute).

For local calls (no binding specified), or references, no interface is required.

Dictionaries

You can use the named parameters notation (`**`) to pass name/value pairs, also known as dictionaries, to/from Python programs. For example:

```
def test(**params):
    for p in params.keys():
        print "key:", p, " parameter:", params[p]
```

and an example of call: `test(a=1, b=2)`

Individual supported types are limited to the types listed in [Table 6-11](#).

Exposing a Python function as an SCA service with the ATMI or Web Services binding requires an interface. This interface is stored in the Tuxedo Service Metadata Repository as outlined in [Python and Ruby Component Programming](#).

For example, consider the Oracle Tuxedo Service Metadata Repository entry shown in [Listing 6-38](#)

Listing 6-38 Oracle Tuxedo Service Metadata Repository Entry for Python

```
##
service=testPython2
tuxservice=TESTPT
inbuf=FML32
outbuf=FML32

param=NUMBER
type=long
access=in

param=TEXT
type=string
access=in
```

```

param=FNUMBER
type=double
access=in
##

```

When called, the Python function receives a list of parameters corresponding exactly to what the interface specifies. Any extra parameters passed by the client are ignored, and any type mismatch results in a data mapping exception.

The names of the parameters match the key names passed to the Python function. The interface is obtained by making an internal call to the [TMMETADATA](#) server. The TMMETADATA server must be running in order to make calls to Python or Ruby functions.

A Python function called with the interface is equivalent to the following Python call:

```
test(a=1, b=2)
```

Ruby Data Type Mapping

In Ruby, clients or components only use parameters and return values types are listed in [Table 6-12](#). Multiple parameters are supported (in the same way that multiple parameters are supported in C++), using FML32 Tuxedo buffers.

Arrays are not supported as they are not supported by bindings or the C++ language extension.

[Table 6-12](#) lists supported Ruby, C/C++ and Tuxedo buffer types.

Table 6-12 Supported Ruby, C++ and Tuxedo Buffer Types

Ruby parameter or return type	C/C++ native type	ATMI binding type
Buffer type/Field type		
Fixnum	short, unsigned short	FML32/FLD_SHORT
Fixnum	long, unsigned long	FML32/FLD_LONG
Bignum	double, long double	FML32/FLD_DOUBLE
True/false	bool	FML32/FLD_CHAR
Float	float	FML32/FLD_FLOAT
Float	double, long double	FML32/FLD_DOUBLE

Table 6-12 Supported Ruby, C++ and Tuxedo Buffer Types

Ruby parameter or return type	C/C++ native type	ATMI binding type
String	char *, std::string	STRING
REXML Object	commonj::sdo::DataObject Ptr	FML32, VIEW, VIEW32

Notes: Ruby runtime may catch an overflow exception.

Possible loss of precision when the Ruby Bignum is bigger than a C++ double.

Float (float) is allowed in C++ to Ruby direction only.

There is no mapping to single character (char/FLD_CHAR) possible in Ruby.

Supported XML objects in Ruby must be REXML (that is, the language extension converts REXML::Document objects into commonj::sdo::DataObject objects, and commonj::sdo::DataObjectPtr objects into REXML::Document objects).

Using variable argument lists and hashes are also supported, as detailed in the following paragraphs.

Note: Variable argument lists and hashes are allowed as parameters, but are not allowed to be returned.

Some limitations concerning multiple parameters and lists will stand with respect to using bindings. For more information, see [Python and Ruby Binding](#).

Ruby Parameters

You can use the list notation (*) to pass an undetermined number of parameters to/from a Ruby script. For example:

```
def func(a, b, *otherargs)
  puts a
  puts b
  otherargs.each { |arg| puts arg }
end
```

which can be called like this: `func(1, 2, 3, 4, 5)`

Individual supported types are limited to the types listed in [Table 6-12](#).

Exposing a Ruby function as an SCA service with the ATMI or Web Services binding requires an interface. This interface is stored in the Tuxedo Service Metadata Repository as outlined in [Python and Ruby Component Programming](#).

For example, consider the Oracle Tuxedo Service Metadata Repository entry shown in [Listing 6-39](#)

Listing 6-39 Oracle Tuxedo Service Metadata Repository Entry for Ruby

```
##
service=testRuby
tuxservice=TESTRU
inbuf=FML32
outbuf=FML32

param=first
type=char
access=in

param=next
type=long
access=in

param=last
type=string
access=in
##
```

When called, the Ruby function receives a list of parameters corresponding exactly to what the interface specifies. Any extra parameters passed by the client are ignored, and any type mismatch results in a data mapping exception.

Notes: Using this notation is limited to local calls (no binding), or with using the ATMI binding between SCA components (that is, the `<binding.atmi>` element with no `requires="legacy"` attribute).

Local calls (no binding specified), or references, do not require an interface.

Hash

You can use named parameters in the form of hash type parameters to pass name/value pairs to/from Ruby scripts. For example:

```
def func2(hash)
  hash.each_pair do |key, val|
    puts "#{key} -> #{val}"
  end
end
```

which can be called like this: `func2("first" => true, "next" => 5, "last" => "hi")`

Individual supported types are limited to the types listed in [Table 6-12](#).

When exposing a Ruby function as an SCA service with the ATMI or Web Services binding, an interface is required. This interface is stored in the Tuxedo Service Metadata Repository as outlined in [Python and Ruby Component Programming](#).

When called, the Ruby function receives a list of parameters corresponding exactly to what the interface specifies. Any extra parameters passed by the client are ignored, and any type mismatch results in a data mapping exception.

The names of the parameters match the key names passed to the Ruby function (that is, a Ruby function called with the above interface is equivalent to the following Ruby client call:

```
func2("first" => true, "next" => 5, "last" => "hi")
```

where the values 'true', 5 and 'hi' are arbitrary, not the keys.

SCA Structure Data Type Mapping

In SCA-ATMI applications, an SCA structure parameter can be mapped to an ATMI FML32, FML, VIEW32, VIEW, X_COMMON, or X_C_TYPE data type, and this is the data type that is specified in the SCA composite file.

If a VIEW32, VIEW, X_COMMON, or X_C_TYPE data type is specified, then this view must exactly match the structure used as an SCA parameter or return type.

Note: In order for the view to exactly match the structure, the compilation of the view needs to produce the same structure with the same fields and same offsets as the structure used in the application.

SCA Structure and FML32 or FML Mapping

If the SCA structure parameter is mapped to FML32 or FML, then the field type of the associated FML32 or FML field is a type that can be converted to and from the SCA structure data type. For more information, see [SCA Data Type Mapping](#).

FML Field Naming Requirements

In SCA-SCA applications, fields are identified by field number, and FML32 field numbers are automatically generated. In the case of nested structures, field numbers are assigned as if the fields in the inner structure had occurred as flat fields in the outer structure in the place where the inner structure is defined in the outer structure.

In SCA-ATMI applications, the FML32 or FML field name associated with a structure element shall be obtained from the structure description file. For more information, see [Using SCA Structure Description Files](#).

Long Element Truncation

When converting an FML32 or FML string, carray, or mbstring field to a structure element, any data that does not fit in the structure element is truncated (without warning) to the provided length.

For example, if a structure element is `char COMPANY_NAME[20]`; and FML field `COMPANY_NAME` with value "International Business Machines" is mapped to this structure element, then "International Busine" is copied to the structure element with no trailing null character.

SCA Structure and VIEW32, VIEW, X_OCTET, or X_C_TYPE Mapping

If an SCA structure is mapped to a VIEW32, VIEW, X_OCTET, or X_C_TYPE data type, then the structure used for the Tuxedo view-based type must exactly match the SCA structure, and is copied byte-by-byte. In other words, no marshalling of data is done when converting between an SCA structure, and a view. FML32 or FML should be used if data marshalling is required.

When an SCA structure is mapped to a view-based Tuxedo type, you cannot specify `bool`, `wchar_t`, `long long`, `unsigned long long`, `long double`, or nested structure data types within the SCA structure since corresponding data types do not exist within Tuxedo views. Elements corresponding to any Tuxedo Associated Count Member or Associated Length Member fields must be provided. Appropriate values for any such elements must also be provided by the application if converting an SCA structure to a Tuxedo view.

SCA Structure and Mbstring Mapping

An mbstring field type currently exists in VIEW32 (for more information, see [tpconvvmb32\(3fm1\)](#)). SCA structures treat the mbstring field type in the same way as in VIEW32. The encoding information is part of an mbstring structure element, and [Fmbunpack32\(\)](#) and [Fmbpack32\(\)](#) must be used in application programs using mbstring data in structures.

TPFAIL Return Data

You can specify a structure pointer as data returned on TPFALL if the same structure pointer is also returned on successful output. Since SCA must store internal information describing the returned structure along with the application data, `<tuxsca.h>` is used to define the structure and typedef as shown in [Listing 6-40](#).

Listing 6-40 `<tuxsca.h>` SCA Structure and Typedef Definition

```
struct scastruct_t {
    void *data;
    void *internalinfo;
};
typedef struct scastruct_t *SCASTRUCT_PTR;
```

If an application normally returns "struct mystruct *" data, it accesses TPFALL data as shown in [Listing 6-40](#).

Listing 6-41 TPFALL Example

```
... catch (Tuscany::sca::atmi::ATMIBindingException& abe) {
    SCASTRUCT_PTR *scap = (SCASTRUCT_PTR *)abe.getData();
    struct mystruct *result = (struct mystruct *)scap->data;
}
```

SCA and Tuxedo Interoperability

Existing Tuxedo service interoperability is performed by using the `/binding.atmi/@requires` attribute with the legacy value. When a legacy value is specified, invocations are performed using the following behavior:

- If a `<map>` element is present in either a `<reference>` or a `<service>`, that value is used to determine which Tuxedo service is associated with the specified method name to call or advertise.

Otherwise:

- In a `<reference>` element: the value specified in the `/reference/@name` element is used to perform the Tuxedo call, with semantics according to the interface method used.
- In a `<service>` element: the Tuxedo service specified in the `/binding.atmi/map` element is advertised, and mapped to the method specified in the `/binding.atmi/map/@target` attribute.

Additionally, the `/binding.atmi/@requires` attribute is used to internally control data mapping, such that FML32 or FML field tables are not required.

Note: When *not* specified, communications are assumed to have SCA -> SCA semantics where the actual Tuxedo service name is constructed from `/service/@name` or `/reference/@name` and actual method name (see the pseudo schema shown [Listing 6-25](#)).

SCA Transactions

The ATMI binding schema supports SCA transaction policies by using the `/binding.atmi/@requires` attribute and three transaction values. These transaction values specify the transactional behavior that the binding extension follows when ATMI binding is used (see the pseudo schema shown [Listing 6-25](#)).

The transaction values are as follows:

- Not specified (no value)

All transactional behavior is left up to the Tuxedo configuration. If the Tuxedo configuration supports transactions, then a transaction can be propagated if it exists.

If the Tuxedo configuration does not support transactions and a transaction exists, then an error will occur.

Note: A transaction is not started if a transaction does not already exist.

- `suspendsTransaction`

When specified, the transaction context is not propagated to the service called. For a `<service>`, the transaction (if present), is automatically suspended before invoking the application code, and resumed afterwards, regardless of the outcome of the invocation. For a `<reference>`, equivalent to making a `tpcall()` with the `TPNOTRAN` flag.

- `propagatesTransaction`

Only applicable to `<reference>` elements, ignored for `<service>` elements. Starts a new transaction if one does not already exist, otherwise participate in existing transaction. Such a behavior can be obtained in a component or composite `<service>` by configuring it `AUTOTRAN` in the `UBBCONFIG`. An error is generated if an Oracle Tuxedo server hosts the SCA component implementation and is not configured in a transactional group in the `UBBCONFIG`.

SCA Security

SCA references pass credentials using the `<authentication>` element of the `binding.atmi` SCDL element.

SCA services can be ACL protected by referencing their internal name:

`/binding.atmi/service/@name` attribute followed by a `'/'` and method name in SCA -> SCA mode, `/binding.atmi/service/@name` attribute in legacy mode (SCA -> Tux interop mode).

See also, [SCA and Tuxedo Interoperability](#).

See Also

- [Oracle SALT Administration Guide](#)
- [Oracle SALT Command Reference Guide](#)
- SDO for C++ Specification V2.1
<http://www.osoa.org/download/attachments/36/Cpp-SDO-Spec-v2.1.0-FINAL.pdf?version=2>
- SCA Assembly Model V0.96:
http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V096.pdf?version=1
- SCA Client and Implementation for C++ (V0.95):

See Also

http://www.osoa.org/download/attachments/35/SCA_ClientAndImplementationModelforCp_p_V0.95.pdf?version=1