gSOAP 2.8.22 User Guide

Robert van Engelen
GENIVIA INC

April 12, 2015

Contents

1 Introduction

The gSOAP tools provide an automated SOAP and XML data binding for C and C++ based on compiler technologies. The tools simplify the development of SOAP/XML Web services and XML application in C and C++ using autocode generation and advanced mapping methods. Most toolkits for Web services adopt a WSDL/SOAP-centric view and offer APIs that require the use of class libraries for XML-specific data structures. This forces a user to adapt the application logic to these libraries because users have to write code to populate XML and extract data from XML using a vendor-specific API. This often leads to fragile solutions with little or no assurances for data consistency, type safety, and XML validation. By contrast, gSOAP provides a type-safe and transparent solution through the use of compiler technology that hides irrelevant WSDL-, SOAP-, REST-, and XML-specific protocol details from the user, while automatically ensuring XML validity checking, memory management, and type-safe serialization. The gSOAP tools automatically map native and user-defined C and C++ data types to semantically equivalent XML data types and vice-versa. As a result, full SOAP/REST XML interoperability is achieved with a simple API relieving the user from the burden of WSDL/SOAP/XML details, thus enabling him or her to concentrate on the application-essential logic.

The gSOAP tools support the integration of (legacy) C/C++ codes (and other programming languages when a C interface is available), embedded systems, and real-time software in SOAP/XML applications that share computational resources and information with other SOAP applications, possibly across different platforms, language environments, and disparate organizations located behind firewalls.

The gSOAP tools are also popular to implement XML data binding in C and C++. This means that application-native data structures can be encoded in XML automatically, without the need to write conversion code. The tools also produce XML schemas for the XML data binding, so external applications can consume the XML data based on the schemas.

1.1 Getting Started

To start building Web services applications or automate XML data bindings with gSOAP, you need:

- The gSOAP package from `http://www.genivia.com/Products/downloads.html` (select gSOAP toolkit standard edition from the list of software packages)

- A C or C++ compiler.

- You may want to install OpenSSL and the Zlib libraries to enable SSL (HTTPS) and compression. These libraries are available for most platforms and are often already installed.

The gSOAP software is self-contained, so there is no need to download any third-party software (unless you want to use OpenSSL and the library is not already installed, or if you need to rebuild the soapcpp2 tool, see below).

The gSOAP packages available from SourceForge include pre-build tools in the gsoap/bin directory:

- The wsdl2h WSDL/schema importer and data binding mapper tool.

- The soapcpp2 stub/skeleton compiler and code generator.

Binaries of these two tools are included in the gSOAP package in gsoap/bin for Windows, Linux, and Mac OS plarforms, see also the README files in the package for more details.

Although gSOAP tools are available in binary format for several platforms, the code generated by these tools are all equivalent. This means that the generated source codes can be transferred to other platforms and locally compiled.

If you don't have the binaries or if you want to rebuild them, you need

- A C compiler and Bison (or Yacc) to build soapcpp2.

- A C compiler and Flex (or Lex) to build soapcpp2.

- A C++ compiler to build wsdl2h.

Bison and Flex are preferred. Both are released under open source licenses that are compatible with gSOAP's licenses.

- Bison is available from `http://www.gnu.org/software/bison`

- Flex is available from `http://flex.sourceforge.net`

The gSOAP engine is built as a library libgsoap.a and libgsoap++.a with separate versions that support SSL. See the README.txt instructions on how to build these libraries with the platform-independent gSOAP package's autoconf and automake. Alternatively, you can compile and link the engine's source code stdsoap2.c (or stdsoap2.cpp for C++) directly with your code.

The gSOAP packages contain numerous examples in the samples directory. Run make to build the example applications. The examples are also meant to demonstrate different features of gSOAP. A streaming MTOM attachment server and client application demonstrate efficient file exchanges in samples/mtom-stream. An SSL-secure Web server application demonstrates the generation of dynamic content for Web browsing and Web services functionality at the same time, see samples/webservice. And much more.

1.2 Quick Start: Developing a Web Service Client Application

The gSOAP tools minimize application adaptation efforts for building Web Services by using a XML data binding for C and C++ implemented by advanced XML schema analyzers and source-to-source code generation tools. The gSOAP wsdl2h tool imports one or more WSDLs and XML schemas and generates a gSOAP header file with familiar C/C++ syntax to define the Web service operations and the C/C++ data types. The gSOAP soapcpp2 compiler then takes this header file and generates XML serializers for the data types (soapH.h and soapC.cpp), the client-side stubs (soapClient.cpp), and server-side skeletons (soapServer.cpp).

The gSOAP soapcpp2 compiler can also generate WSDL definitions for implementing a service from scratch, i.e. without defining a WSDL first. This "closes the loop" in that it enables Web services

development from WSDL or directly from a set op C/C++ operations in a header file without the need for users to analyze Web service details.

You only need to follow a few steps to execute the tools from the command line or Makefile (see also MSVC++ project examples in the samples directory with tool integration in the MSVC++ IDE). For example, to generate code for the calculator Web service, we run the wsdl2h tool from the command line on the URL of the WSDL and use option -o to specify the output file:

```
> wsdl2h -o calc.h http://www.genivia.com/calc.wsdl
```

This generates the calc.h service definition header file with service operation definitions and types for the operation's data. This header file is then to be processed with soapcpp2 to generate the stub and/or skeleton code and XML serialization routines. The calc.h file includes all documentation, so you can use Doxygen (`http://www.doxygen.org`) to automatically generate the documentation pages for your development.

The wsdl2h-generated service definitions header file also contains information on the use of the service, such as WS-Policy assertions when applicable.

In this example we are developing a C++ API for the calculator service. By default, gSOAP assumes you will use C++ with STL. To build without STL, use option -s:

```
> wsdl2h -s -o calc.h http://www.genivia.com/calc.wsdl
```

To build a pure C application, use option -c:

```
> wsdl2h -c -o calc.h http://www.genivia.com/calc.wsdl
```

Important: Visual Studio users shopuld make sure to compile all gSOAP source files in C++ compilation mode. If you migrate to a project file .vcproj, please set `CompileAs="2"` in your .vcproj file. We have not yet generated the stubs for the C/C++ API. To do so, run the soapcpp2 compiler:

```
> soapcpp2 -i -C -Iimport calc.h
```

Option -i (and alternatively option -j) indicates that we want C++ proxy and server objects that include the client (and server) code, -C indicates client-side only files (soapcpp2 generates both client and server stubs and skeletons by default). Option -I is needed to import the stlvector.h file from the import directory in the gSOAP package to support serialization of STL vectors.

Suppose we develop a C++ client for the calculator service using wsdl2h -o calc.h http://www.genivia.com/calc.wsdl and soapcpp2 -i -C calc.h.

We use the generated soapcalcProxy class and calc.nsmap XML namespace mapping table to access the Web service. The soapcalcProxy class is a proxy to invoke the service:

```cpp
#include "soapcalcProxy.h"
#include "calc.nsmap"
int main()
{
    calcProxy service;
```

```
    double result;
    if (service.add(1.0, 2.0, result) == SOAP_OK)
       std::cout << "The sum of 1.0 and 2.0 is " << result << std::endl;
    else
       service.soap_stream_fault(std::cerr);
    service.destroy(); // delete data and release memory
}
```

To complete the build, compile the code above and compile and link this with the generated soapC.cpp, soapcalcProxy.cpp, and the run-time gSOAP engine -lgsoap++ (or use source stdsoap2.cpp in case libgsoap++.a is not installed) with your code.

Suppose we develop a client in C using wsdl2h -c -o calc.h http://www.genivia.com/calc.wsdl and soapcpp2 -C calc.h. In this case our code uses a simple C function call to invoke the service and we also need to explicitly delete data and the context with soap_end and soap_free:

```
#include "soapH.h"
#include "calc.nsmap"
int main()
{
    struct soap *soap = soap_new();
    double result;
    if (soap_call_ns__add(soap, 1.0, 2.0, &result) == SOAP_OK)
       printf("The sum of 1.0 and 2.0 is %lg\n", result);
    else
       soap_print_fault(soap, stderr);
    soap_end(soap);
    soap_free(soap);
}
```

The calculator example is fairly simple and used here to illustrate the development process. The development process for large-scale XML applications is similar. More extensive examples can be found in the samples directory in the gSOAP package.

1.3   Quick Start: Developing a Web Service

Developing a service application is easy too. We will use CGI here because it is a simple mechanism. This is not the preferred deployment mechanism. Because CGI is slow and stateless. Faster services can be developed as a stand-alone gSOAP HTTP/HTTPS server (but see comments at the end of this section) or, as preferred, the use of Apache module or IIS (included in the gSOAP package under gsoap/mod_gsoap with instructions).

Suppose we implement a CGI-based service that returns the time in GMT. The Common Gateway Interface (CGI) is a simple mechanism to publish services on your Web site.

For this example we start with a gSOAP header file, currentTime.h which contains the service definitions. Such a file can be obtained from a WSDL using wsdl2h when a WSDL is available. When a WSDL is not available, you can define the service in C/C++ definitions in a newly created header file and let the gSOAP tools generate the source code and WSDL for you.

Our currentTime service only has an output parameter, which is the current time defined in our currentTime.h gSOAP service specification:

```
// File: currentTime.h
//gsoap ns service name: currentTime
//gsoap ns service namespace: urn:currentTime
//gsoap ns service location: http://www.yourdomain.com/currentTime.cgi
int ns__currentTime(time_t& response);
```

Note that we associate an XML namespace prefix ns and namespace name urn:currentTime with the service WSDL and SOAP/XML messages. The gSOAP tools use a special convention for identifier names that are part of a namespace: a namespace prefix (ns in this case) followed by a double underscore __. This convention is used to resolve namespaces and to avoid name clashes. The ns namespace prefix is bound to the urn:currentTime namespace name with the //gsoap directive. The //gsoap directives are used to set the properties of the service, in this case the name, namespace, and location endpoint.

The service implementation for CGI requires a soap_serve call on a soap context created with soap_new. The service operations are implemented as functions, which are called by the RPC dispatcher soap_serve:

```
// File: currentTime.cpp
#include "soapH.h" // include the generated declarations
#include "currentTime.nsmap" // include the XML namespace mappings
int main()
{
   // create soap context and serve one CGI-based request:
   return soap_serve(soap_new());
}
int ns__currentTime(struct soap *soap, time_t& response)
{
   response = time(0);
   return SOAP_OK;
}
```

Note that we pass the soap struct with the gSOAP context information to the service routine. This can come in handy to determine properties of the connection and to dynamically allocate data with soap_malloc(soap, num_bytes) that will be automatically deleted when the service is finished.

We run the soapcpp2 compiler on the header file to generate the server-side code:

```
> soapcpp2 -S currentTime.h
```

and then compile the CGI binary:

```
> c++ -o currentTime.cgi currentTime.cpp soapC.cpp soapServer.cpp stdsoap2.cpp
```

You will find stdsoap2.cpp in the gsoap dir. Or after installation you can link with libgsoap++ instead of using the stdsoap2.cpp source:

```
> c++ -o currentTime.cgi currentTime.cpp soapC.cpp soapServer.cpp -lgsoap++
```

To activate the service, copy the currentTime.cgi binary to your bin-cgi directory using the proper file permissions.

The soapcpp2 tool generated the WSDL definitions currentTime.wsdl. You can use the WSDL to advertize your service. You don't need to use this WSDL to develop a gSOAP client. You can use the currentTime.h file with soapcpp2 option -C to generate client-side code.

A convenient aspect of CGI is that it exchanges messages over standard input/output. Therefore, you can run the CGI binary on the auto-generated example request XML file to test your server:

```
> ./currentTime.cgi < currentTime.currentTime.req.xml
```

and this displays the server response in SOAP XML.

The above approach works also for C. Just use soapcpp2 option -c in addition to the -S option to generate ANSI C code. Of course, in C we use pointers instead of references and the currentTime.h file should be adjusted to use C-only types.

A more elegant server implementation in C++ can be obtained by using the soapcpp2 option -i (or -j) to generate C++ client-side proxy and server-side service objects as classes that you can use to build clients and services in C++. The option removes the generation of soapClient.cpp and soapServer.cpp, since these are no longer needed when we have classes that implement the client and server logic:

```
> soapcpp2 -i -S currentTime.h
```

This generates soapcurrentTimeService.h and soapcurrentTimeService.cpp files, as well as auxiliary files soapStub.h (included by default by all codes) and currentTime.nsmap.

Using the currentTimeService object we rewrite the CGI server as:

```
// File: currentTime.cpp
#include "soapcurrentTimeService.h" // include the proxy declarations
#include "currentTime.nsmap" // include the XML namespace mappings
int main()
{
  // create server and serve one CGI-based request:
  currentTimeService server;
  server.serve();
  server.destroy();
}
int currentTimeService::currentTime(time_t& response)
{
  response = time(0);
  return SOAP_OK;
}
```

Compile with

```
> c++ -o currentTime.cgi currentTime.cpp soapC.cpp soapcurrentTimeService.cpp -lgsoap++
```

and install the binary as CGI. To install the CGI binary please consult your Web server documentation on how to deploy CGI applications.

To run the server as a stand-alone iterative (non-multithreaded) server on port 8080:

```
while (server.run(8080) != SOAP_TCP_ERROR)
    server.soap_stream_fault(std::cerr);
```

To implement threaded services, please see Section 7.2.4. These stand-alone Web Services that handle multiple SOAP requests by spawning a thread for each request. Thread pooling is also an option. The use of Apache and IIS modules to deploy gSOAP services is preferred to ensure load balancing, access control, tracing, and so on.

For more information on server-side service classes, see Section 7.2.7. For more information on client-side proxy classes, see Section 7.1.4.

1.4 Quick Start: XML Data Bindings

Or in other words, how to map schemas (XSD files) to C/C++ bindings for automatically reading and writing XML data.

The XML C/C++ data binding in gSOAP provides and automated mechanism to encode any C and C++ data type in XML (and decode XML back to C/C++ data). An XML schema (XSD file) can be transformed into a set of C or C++ definitions that can be readily incorporated into your application to manipulate XML with much more ease than DOM or SAX. Essentially, each XML component definition in an XML schema has a C/C++ data type representation that has equivalent type properties. The advantage of this approach is immediately apparent when we look at an XSD complexType that maps to a class:

```
XSD                                                       C++
<complexType name="Address">                              class ns__Address
  <sequence>                                              {
    <element name="name" type="string"/>                   std::string name;
    <element name="home" type="tns:Location" minOccurs="0"/>  ns__Location *home;
    <element name="phone" type="unsignedLong"/>            ULONG64 phone;
    <element name="dob" type="dateTime"/>                  time_t dob;
  </sequence>
  <attribute name="ID" type="int" use="required"/>         @int ID;
</complexType>                                             }
```

In this way, an automatic mapping between XML elements of the XML schema and members of a class is created to No DOM traversals and SAX events are needed. In addition, the XML C/C++ data binding makes XML manipulation type safe. That is, the type safety of C/C++ ensures that only valid XML documents can be parsed and generated.

The wsdl2h tool performs the mapping of WSDL and XML schemas to C and/or C++ automatically. The output of wsdl2h is an annotated header file that includes comments and documentation on the use of the C/C++ declarations in a SOAP/XML client/server or in a generic application for reading and writing XML using the auto-generated serializers.

We will illustrate this further with an example. Suppose we have an XML document with a book record:

```
<book isbn="1234567890">
  <title>Farewell John Doe</title>
  <publisher>ABC's is our Name</publisher>
</book>
```

An example XML schema that defines the book element and type could be

```
<schema ...>
  <element name="book">
    <complexType>
      <sequence>
        <element name="title" type="string" minOccurs="1"/>
        <element name="publisher" type="string" minOccurs="1"/>
      </sequence>
      <attribute name="isbn" type="unsignedLong" use="required"/>
    </complexType>
  </element>
</schema>
```

Using wsdl2h we translate the XML schema that defines the book type and root element to a class definition:

```
class book
{
  @ULONG64 isbn;
  std::string title;
  std::string publisher;
}
```

Note that annotations such as @ are used to distinguish attributes from elements. These annotations are gSOAP-specific and are handled by the soapcpp2 tool to generate serializers for the data type(s) for reading and writing XML.

The soapcpp2 tool generates all the necessary code to parse and generate XML for book objects. Validation constraints such as `minOccurs="1"` and `use="required"` are included in the generated code as checks.

To write the XML representation of a book, we first create a soap engine context and use it with soap_write_book (generated by soapcpp2) to write the object in XML to standard output:

```
soap *ctx = soap_new1(SOAP_XML_INDENT); // new context with option
book bk;
bk.isbn = 1234567890;
bk.title = "Farewell John Doe";
bk.publisher = "ABC's is our Name";
if (soap_write_book(ctx, &bk) != SOAP_OK)
  ... error ...
soap_destroy(ctx); // clean up allocated class instances
```

```
soap_end(ctx); // clean up allocated temporaries
soap_free(ctx); // delete context
```

The ctx gSOAP engine context (type **struct** soap) controls settings and holds state, such as XML formatting, (e.g. SOAP_XML_INDENT), serialization options, current state, and I/O settings. Simply set the output stream (std::ostream) ctx->os of the context to redirect the content, e.g. to a file or string. Also, when serializing a graph rather than an XML tree (SOAP_XML_TREE option forces trees) the XML output conforms to SOAP encoding for object graphs based on id-ref, see Section 7.5 for details.

To read the XML representation from standard input into a book object, use:

```
soap *ctx = soap_new1(SOAP_XML_STRICT); // new context with option
book bk;
if (soap_read_book(ctx, &bk) != SOAP_OK)    ... error ...
else
   cout ¡¡ bk.isbn ¡¡ ", " ¡¡ bk.title ¡¡ ", " ¡¡ bk.publisher ¡¡ endl;
... further use of bk ...
soap_destroy(ctx); // delete deserialized objects
soap_end(ctx); // delete temporaries
soap_free(ctx); // delete context
```

Automatic built-in XML validation (enabled with SOAP_XML_STRICT) ensures that data members are present so we can safely print them in this example, thus ensuring consistency of data with the XML schema. Set the ctx->is input stream to read from a file/string stream instead of stdin.

The soap_destroy and soap_end calls deallocate the deserialized content, so use with care. In general, memory management is automatic in gSOAP to avoid leaks.

The above uses a very simple example schema. The gSOAP toolkit handles all XML schema constructs defined by the XML schema standard. The toolkit is also able to (de)serialize pointer-based C/C++ data structures (including cyclic graphs), structs/classes, unions, enums, STL containers, and even special data types such as struct tm. Therefore, the toolkit works in two directions: from WSDL/schema to C/C++ and from C/C++ to WSDL/schema.

The gSOAP toolkit also handles multiple schemas defined in multiple namespaces. Normally the namespace prefixes of XML namespaces are added to the C/C++ type definitions to ensure type uniqueness. For example, if we would combine two schemas in the same application where both schemas define a book object, we need to resolve this conflict. In gSOAP this is done using namespace prefixes, rather than C++ namespaces (research has pointed out that XML namespaces are not equivalent to C++ namespaces). Thus, the book class might actually be bound to an XML namespace and the class would be named ns__book, where ns is bound to the corresponding namespace.

The following options are available to control serialization:

```
soap-¿encodingStyle = NULL; // to remove SOAP 1.1/1.2 encodingStyle
soap_mode(soap, SOAP_XML_TREE); // XML without id-ref (no cycles!)
soap_mode(soap, SOAP_XML_GRAPH); // XML with id-ref (including cycles)
soap_set_namespaces(soap, struct Namespace *nsmap); //to set xmlns bindings
```

Other flags can be used to format XML, see Section 9.12.

More information on XML databinding support for C and C++, see Section 7.5.

1.5   Feature Overview

The highlights of gSOAP are:

- Unique interoperability features: the tools generate type-safe SOAP marshalling routines to (de)serialize native and user-defined C and C++ data structures.

- Support WSDL 1.1, WSDL 2.0, REST, SOAP 1.1, SOAP 1.2, SOAP RPC encoding style, and document/literal style. gSOAP is one of the few SOAP toolkits that support the **full** range of SOAP 1.1 RPC encoding features including sparse multi-dimensional arrays and polymorphic types. For example, a service operation with a base class parameter may accept derived class instances from a client. Derived class instances keep their identity through dynamic binding. The toolkit also supports **all XSD 1.0 and 1.1 schema type** constructs and has been tested against the W3C XML Schema Patterns for Databinding Interoperability working group and of gSOAP release 2.8.x passes all tests.

- Supports WS-Security, WS-Addressing, WS-ReliableMessaging, C14N exclusive canonicalization. The protocols are implemented using code generation with wsdl2h and soapcpp2. The gSOAP tools can be used to generate messaging protocols for other WS-* protocols.

- gSOAP supports XML-RPC and RSS protocols. Examples are provided.

- JSON support is included in the XML-RPC library to switch between XML-RPC and JSON protocols. For more details, see the samples/xml-rpc-json folder in the package.

- The wsdl2h tool supports WS-Policy. Policy assertions are included in the generated service description header file with recommendations and usage hints.

- gSOAP supports MIME (SwA), DIME, and MTOM attachments and has streaming capabilities to direct the data stream to/from resources. gSOAP is the only toolkit that supports *streaming* MIME, DIME, and MTOM attachment transfers, which allows you to exchange binary data of practically unlimited size in the fastest possible way (streaming) while ensuring the usefulness of XML interoperability.

- gSOAP supports SOAP-over-UDP.

- gSOAP supports IPv4 and IPv6.

- gSOAP supports Zlib deflate and gzip compression (for HTTP, TCP/IP, and XML file storage).

- gSOAP supports SSL (HTTPS) using OpenSSL and optionally using GNUTLS.

- gSOAP supports HTTP/1.0, HTTP/1.1 keep-alive, chunking, basic authentication, and digest authentication using a plugin.

- gSOAP supports SOAP one-way messaging.

- The schema-specific XML pull parser is fast and efficient and does not require intermediate data storage for demarshalling to save space and time.

- The soapcpp2 compiler includes a WSDL and schema generator for convenient Web Service publishing.

- The soapcpp2 compiler generates sample input and output messages for verification and testing (before writing any code). An option (-T) can be used to automatically implement echo message services for testing.

- The wsdl2h tool converts WSDL and XSD files to gSOAP header files for automated client and server development.

- Generates source code for stand-alone Web Services and client applications.

- Ideal for small devices such as Palm OS, Symbian, Pocket PC, because the memory footprint is small.

- Ideal for building web services that are compute-intensive and are therefore best written in C and C++.

- Platform independent: Windows, Unix, Linux, Mac OS X, Pocket PC, Palm OS, Symbian, VXWorks, etc.

- Supports serializing of application's native C and C++ data structures, which allows you to save and load XML serialized data structures to and from files.

- Selective input and output buffering is used to increase efficiency, but full message buffering to determine HTTP message length is not used. Instead, a three-phase serialization method is used to determine message length. As a result, large data sets such as base64-encoded images can be transmitted with or without DIME attachments by small-memory devices such as PDAs.

- Supports C++ single class inheritance, dynamic binding, overloading, arbitrary pointer structures such as lists, trees, graphs, cyclic graphs, fixed-size arrays, (multi-dimensional) dynamic arrays, enumerations, built-in XSD Schema types including base64Binary encoding, and hexBinary encoding.

- No need to rewrite existing C/C++ applications for Web service deployment. However, parts of an application that use unions, pointers to sequences of elements in memory, and **void*** need to be modified, but **only** if the data structures that adopt them are required to be serialized or deserialized as part of a service operation invocation.

- Three-phase marshalling: 1) analysis of pointers, single-reference, multi-reference, and cyclic data structures, 2) HTTP message-length determination, and 3) serialization as per SOAP 1.1 encoding style or user-defined encoding styles.

- Two-phase demarshalling: 1) SOAP parsing and decoding, which involves the reconstruction of multi-reference and cyclic data structures from the payload, and 2) resolution of "forward" pointers (i.e. resolution of the forward `href` attributes in SOAP).

- Full and customizable SOAP Fault processing (client receive and service send).

- Customizable SOAP Header processing (send and receive), which for example enables easy transaction processing for the service to keep state information.

2   Notational Conventions

The typographical conventions used by this document are:

Sans serif or italics font denotes C and C++ source code, file names, and shell/batch commands.

Bold font denotes C and C++ keywords.

`Courier font` denotes HTTP header content, HTML, XML, XML Schema content and WSDL content.

[Optional] denotes an optional construct.

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119.

3   Differences Between gSOAP Versions 2.4 (and Earlier) and 2.5

To comply with WS-I Basic Profile 1.0a, gSOAP 2.5 and higher adopts SOAP document/literal by default. There is no need for concern, because the WSDL parser wsdl2h automatically takes care of the differences when you provide a WSDL document, because SOAP RPC encoding, literal, and document style are supported. A new soapcpp2 compiler option was added -e for backward compatibility with gSOAP 2.4 and earlier to adopt SOAP RPC encoding by default in case you want to develop a service that uses SOAP encoding. You can also use the gSOAP soapcpp2 compiler directives to specify SOAP encoding for individual operarations, when desired.

4   Differences Between gSOAP Versions 2.1 (and Earlier) and 2.2

You should read this section only if you are upgrading from gSOAP 2.1 to 2.2 and later.

Run-time options and flags have been changed to enable separate recv/send settings for transport, content encodings, and mappings. The flags are divided into four classes: transport (IO), content encoding (ENC), XML marshalling (XML), and C/C++ data mapping (C). The old-style flags soap_disable_X and soap_enable_X, where X is a particular feature, are deprecated. See Section 9.12 for more details.

5   Differences Between gSOAP Versions 1.X and 2.X

You should read this section only if you are upgrading from gSOAP 1.X to 2.X.

gSOAP versions 2.0 and later have been rewritten based on versions 1.X. gSOAP 2.0 and later is thread-safe, while 1.X is not. All files in the gSOAP 2.X distribution are renamed to avoid confusion with gSOAP version 1.X files:

gSOAP 1.X	gSOAP 2.X
soapcpp	soapcpp2
soapcpp.exe	soapcpp2.exe
stdsoap.h	stdsoap2.h
stdsoap.c	stdsoap2.c
stdsoap.cpp	stdsoap2.cpp

Changing the version 1.X application codes to accommodate gSOAP 2.X does not require a significant amount of recoding. The change to gSOAP 2.X affects all functions defined in stdsoap2.c[pp] (the gSOAP runtime context API) and the functions in the sources generated by the gSOAP soapcpp2 compiler (the gSOAP RPC+marshalling API). Therefore, clients and services developed with gSOAP 1.X need to be modified to accommodate a change in the calling convention used in 2.X: In 2.X, **all** gSOAP functions (including the service operation proxy routines) take an additional parameter which is an instance of the gSOAP runtime context that includes file descriptors, tables, buffers, and flags. This additional parameter is **always** the first parameter of any gSOAP function.

The gSOAP runtime context is stored in a **struct** soap type. A **struct** was chosen to support application development in C without the need for a separate gSOAP implementation. An object-oriented approach with a class for the gSOAP runtime context would have prohibited the implementation of pure C applications. Before a client can invoke service operations or before a service can accept requests, a runtime context needs to be allocated and initialized. Three new functions are added to gSOAP 2.X:

Function	Description
soap_init(**struct** soap *soap)	Initializes a context (required only once)
struct soap *soap_new()	Allocates, initializes, and returns a pointer to a runtime context
struct soap *soap_copy(**struct** soap *soap)	Allocates a new runtime context and copies contents of the context such that the new environment does not share any data with the original context

A context can be reused as many times as necessary and does not need to be reinitialized in doing so. A dynamically allocated context is deallocated with soap_free.

A new context is only required for each new thread to guarantee exclusive access to a new runtime context by each thread. For example, the following code stack-allocates the runtime context which is used for multiple service operation calls:

```
int main()
{
```

```
struct soap soap;
...
soap_init(&soap); // initialize runtime context
...
soap_call_ns__method1(&soap, ...); // make a remote call
...
soap_call_ns__method2(&soap, ...); // make another remote call
...
soap_destroy(&soap); // remove deserialized class instances (C++ only)
soap_end(&soap); // clean up and remove deserialized data
soap_done(&soap); // detach context (last use and no longer in scope)
...
}
```

The runtime context can also be heap allocated:

```
int main()
{
  struct soap *soap;
  ...
  soap = soap_new(); // allocate and initialize runtime context
  if (!soap) // couldn't allocate: stop
  ...
  soap_call_ns__method1(soap, ...); // make a remote call
  ...
  soap_call_ns__method2(soap, ...); // make another remote call
  ...
  soap_destroy(soap); // remove deserialized class instances (C++ only)
  soap_end(soap); // clean up and remove deserialized data
  soap_free(soap); // detach and free runtime context
}
```

A service needs to allocate and initialize an context before calling soap_serve:

```
int main()
{
  struct soap soap;
  soap_init(&soap);
  soap_serve(&soap);
}
```

Or alternatively:

```
int main()
{
  soap_serve(soap_new());
}
```

The soap_serve dispatcher handles one request or multiple requests when HTTP keep-alive is enabled (with the SOAP_IO_KEEPALIVE flag see Section 19.11).

A service can use multi-threading to handle requests while running some other code that invokes service operations:

```
int main()
{
    struct soap soap1, soap2;
    pthread_t tid;
    ...
    soap_init(&soap1);
    if (soap_bind(&soap1, host, port, backlog) < 0) exit(1);
    if (soap_accept(&soap1) < 0) exit(1);
    pthread_create(&tid, NULL, (void*(*)(void*))soap_serve, (void*)&soap1);
    ...
    soap_init(&soap2);
    soap_call_ns_ _method(&soap2, ...); // make a remote call
    ...
    soap_end(&soap2);
    ...
    pthread_join(tid, NULL); // wait for thread to terminate
    soap_end(&soap1); // release its data
}
```

In the example above, two runtime contexts are required. In comparison, gSOAP 1.X statically allocates the runtime context, which prohibits multi-threading (only one thread can invoke service operations and/or accept requests due to the single runtime context).

Section 7.2.4 presents a multi-threaded stand-alone Web Service that handles multiple SOAP requests by spawning a thread for each request.

6 Interoperability

gSOAP interoperability has been verified with the following SOAP implementations and toolkits:

Apache 2.2
Apache Axis
ASP.NET
Cape Connect
Delphi
easySOAP++
eSOAP
Frontier
GLUE
Iona XMLBus
kSOAP
MS SOAP
Phalanx
SIM

SOAP::Lite
SOAP4R
Spray
SQLData
WCF
White Mesa
xSOAP
ZSI
4S4C

7 Quick User Guide

This user guide offers a quick way to get started with gSOAP. This section requires a basic understanding of the SOAP protocol and some familiarity with C and/or C++. In principle, SOAP clients and SOAP Web services can be developed in C and C++ with the gSOAP soapcpp2 compiler without a detailed understanding of the SOAP protocol when gSOAP client-server applications are built as an ensamble and only communicate within this group (i.e. meaning that you don't have to worry about interoperability with other SOAP implementations). This section is intended to illustrate the implementation of gSOAP Web services and clients that connect to and interoperate with other SOAP implementations such as Apache Axis, SOAP::Lite, and .NET. This requires some details of the SOAP and WSDL protocols to be understood.

7.1 How to Build SOAP/XML Clients

In general, the implementation of a SOAP client application requires a *stub* (also called *service proxy*) for each service operation that the client invokes. The primary stub's responsibility is to marshall the parameter data, send the request with the parameters to the designated SOAP service over the wire, to wait for the response, and to demarshall the parameter data of the response when it arrives. The client application invokes the stub routine for a service operation as if it would invoke a local function. To write a stub routine in C or C++ by hand is a tedious task, especially if the input and/or output parameters of a service operation contain elaborate data structures such as objects, structs, containers, arrays, and pointer-linked graph structures. Fortunately, the gSOAP wsdl2h WSDL parser tool and soapcpp2 stub/skeleton and serialization code generator tool automate the development of SOAP/XML Web service client and server applications.

The soapcpp2 tool generates the necessary gluing code (also called stubs and skeletons) to build web service clients and services. The input to the soapcpp2 tool consists of an service definition-annotated C/C++ **header file**. The header file can be generated from a WSDL (Web Service Description Language) documentation of a service with the gSOAP wsdl2h WSDL parser tool.

Consider the following command (entered at the Linux/Unix/Windows command line prompt):

> wsdl2h -o calc.h http://www.genivia.com/calc.wsdl

This generates the file Web service description calc.h in an annotated C++ header file. The WSDL

specification (possibly consisting of multiple imported WSDL files and XSD schema files) is mapped to C++ using C++ databindings for SOAP/XML. The generated header file contains data types and messages to operate the service, and meta information related to WSDL and XML schemas.

To generate a service definition header file to develop a pure C client application, use the -c option:

> wsdl2h -c -o calc.h http://www.genivia.com/calc.wsdl

For more details on the WSDL parser and its options, see 8.

The service definition calc.h header file is further processed by the gSOAP soapcpp2 compiler to generate the gluing code's logic to invoke the Web service from a client.

Looking into the file calc.h we see that the SOAP service methods are specified as **function prototypes**. For example, the add function to add two double floats:

int ns2_ _add(**double** a, **double** b, **double**& result);

The ns2_ _add function uses an XML namespace prefix to distinguish it from operations defined in other namespaces, thus preventing name clashes. The convention to add an XML namespace prefix to the names of operations, types, and **struct** and **class** members is universally used by the gSOAP tools and automatically created by wsdl2h, but it is not mandatory when translating existing C/C++ types and operations to web services. Thus, the prefix notation can be omitted from type names defined in an header file with to run soapcpp2 to create clients and services that exchange existing (i.e. application-native) data types.

These function prototypes are translated by the gSOAP soapcpp2 tool to stubs and proxies for remote calls:

soapStub.h	annotated copy of the input definitions
soapH.h	serializers
soapC.cpp	serializers
soapClient.cpp	client calling stubs

Thus, the logic of the generated stub routines allow C and C++ client applications to seamlessly interact with existing SOAP Web services as illustrated by the client code example in the next section.

The input and output parameters of a SOAP service operation may be primitive data types or complex compound data types such as containers and pointer-based linked data structures. These are defined in the header file that is either generated by the WSDL parser or specified by hand. The gSOAP soapcpp2 tool automatically generates **XML serializers** and **XML deserializers** for the data types to enable the generated stub routines to encode and decode the contents of the parameters of the service operations in SOAP/XML.

Note that the gSOAP soapcpp2 tool also generates **skeleton** routines soapServer.cpp for each of the service operations specified in the header file. The skeleton routines can be readily used to implement one or more of the service operations in a new SOAP Web service. These skeleton routines are not used for building SOAP clients in C++, although they can be used to build mixed SOAP client/server applications (peer applications).

7.1.1 Example

The `add` service operation (declared in the `calc.h` file obtained with the `wsdl2h` tool in the previous section) adds two float values. The WSDL description of the service provides the endpoint to invoke the service operations and the XML namespace used by the operations:

    Endpoint URL: `http://websrv.cs.fsu.edu/ engelen/calcserver.cgi`
    XML namespace: `urn:calc`

Each service operation has a SOAP action, which is an optional string to identify the operation (mainly used with WS-Addressing), an operation request message and a response message. The request and response messages for SOAP RPC-encoded services are simply represented by C functions with input and output parameters. For the `add` operation, the SOAP binding details are:

    SOAP style: RPC
    SOAP encoding: encoded
    SOAP action: `""` (empty string)

This information is translated to the `wsdl2h`-generated header file with the service definitions. The `calc.h` header file for C++ generated by `wsdl2h` contains the following directives and declarations: (the actual contents may vary depending on the release version and the options used to control the output):

```
//gsoap ns2 service name: calc //gsoap ns2 service type: calcPortType //gsoap ns2 service port:
http://websrv.cs.fsu.edu/ engelen/calcserver.cgi
//gsoap ns2 service namespace: urn:calc

//gsoap ns2 service method-protocol: add SOAP
//gsoap ns2 service method-style: add rpc
//gsoap ns2 service method-encoding: add http://schemas.xmlsoap.org/soap/encoding/
//gsoap ns2 service method-action: add ""
int ns2__add(double a, double b, double& result);
```

The other calculator operations are similar and elided here for clarity.

The `//gsoap` directives are required for the `soapcpp2` tool to generate code that is compliant to the SOAP protocol. For this service the SOAP protocol with the common "RPC encoding style" is used. For `//gsoap` directive details, see Section 19.2.

The service operations are declared as function prototypes, with all non-primitive parameter types needed by the operation declared in the header file (all parameter types are primitive in this case).

The calculator `add` opertion takes two double floats `a` and `b`, and returns the sum in `result`. By convention, **all parameters are input parameters except the last**. The last parameter is always the output parameter. A **struct** or **class** is used to wrap multiple output parameters, see also Section 7.1.9. This last parameter must be a pointer or reference. By contrast, the input parameters support pass by value or by pointer, but not pass by C++ reference.

The function prototype associated with a service operation always returns an **int**. The value indicates success (`0` or equivalently `SOAP_OK`) or failure (any nonzero value). See Section 10.2 for the nonzero error codes.

The role of the namespace prefix (ns2__) in the service operation name in the function prototype declaration is discussed in detail in 7.1.2. Basically, a namespace prefix is added to a function name or type name with a **pair of underscores**, as in ns2__add, where ns2 is the namespace prefix and add is the service operation name. This mechanism ensures uniqueness of operations and types associated with a service.

It is strongly recommended to set the namespace prefix to a name of your choice. This avoids problems when running wsdl2h on multiple WSDLs where the sequence of prefixes ns1, ns2, and so on are arbitrarily assigned to the services. To choose a prefix name for all the operations and types of a service, say prefix c__ for the calculator service, add the following line to typemap.dat:

```
c = "urn:calc"
```

and rerun wsdl2h. The typemap.dat configures wsdl2h to use specific bindings and data types for services. The result is that c__add is used to uniquely identify the operation rather than the more arbitrary name ns2__add.

Note on the use of underscores in names: a single underscore in an identifier name will be translated into a dash in XML, because dashes are more frequently used in XML compared to underscores, see Section 10.3.

Next, the gSOAP soapcpp2 tool is invoked from the command line to process the calc.h service definitions:

```
> soapcpp2 calc.h
```

The tool generates the stub routines for the service operations. Stub routines can be invoked by a client program to invoke the remote service operations. The interface of the generated stub routine is identical to the function prototype in the calc.h service defintion file, but with additional parameters to pass the gSOAP engine's runtime context soap, an endpoint URL (or NULL for the default), and a SOAP action (or NULL for the default):

int soap_call_c__add(**struct** soap *soap, **char** *URL, **char** *action, **double** a, **double** b, **double**& result);

This stub routine is saved in soapClient.cpp. The file soapC.cpp contains the **serializer** and **deserializer** routines for the data types used by the stub. You can use option -c for the soapcpp2 tool to generate pure C code, when needed.

Note: the soap parameter must be a valid pointer to a gSOAP runtime context. The URL can be set to override the default endpoint address (the endpoint defined by the WSDL). The action parameter can be set to override the default SOAP action.

The following example C/C++ client program uses the stub:

```
#include "soapH.h" // include all interfaces (library and generated)
#include "calc.nsmap" // import the generated namespace mapping table
int main()
{
    double sum;
```

```
    struct soap soap; // the gSOAP runtime context
    soap_init(&soap); // initialize the context (only once!)
    if (soap_call_c__add(&soap, NULL, NULL, 1.0, 2.0, &sum) == SOAP_OK)
        std::cout << "Sum = " << sum << std::endl;
    else // an error occurred
        soap_print_fault(&soap, stderr); // display the SOAP fault message on the stderr stream
    soap_destroy(&soap); // delete deserialized class instances (for C++)
    soap_end(&soap); // remove deserialized data and clean up
    soap_done(&soap); // detach the gSOAP context
    return 0;
}
```

The call returns SOAP_OK (zero) on success and a nonzero error on failure. When an error occurred the fault is displayed with the soap_print_fault function. Use soap_sprint_fault(**struct** soap*, **char** *buf, size_t len) to print the error to a string, and use soap_stream_fault(**struct** soap*, std::ostream&) to send it to a stream (C++ only).

The following functions can be used to explicitly setup a gSOAP runtime context (**struct** soap):

Function	Description
soap_init(**struct** soap *soap)	Initializes a runtime context
soap_init1(**struct** soap *soap, soap_mode iomode)	Initializes a runtime context and set in/out mode flags
soap_init2(**struct** soap *soap, soap_mode imode, soap_mode omode)	Initializes a runtime context and set in/out mode flags
struct soap *soap_new()	Allocates, initializes, and returns a pointer to a runtime context
struct soap *soap_new1(soap_mode iomode)	Allocates, initializes, and returns a pointer to a runtime context and set in/out mode flags
struct soap *soap_new2(soap_mode imode, soap_mode omode)	Allocates, initializes, and returns a pointer to a runtime context and set in/out mode flags
struct soap *soap_copy(**struct** soap *soap)	Allocates a new runtime context and copies a context (deep copy, i.e. the new context does not share any data with the other context)
soap_done(**struct** soap *soap)	Reset, close communications, and remove callbacks
soap_free(**struct** soap *soap)	Reset and deallocate the context created with soap_new or soap_copy

A runtime context can be reused as many times as necessary for client-side remote calls and does not need to be reinitialized in doing so. A new context is required for each new thread to guarantee exclusive access to runtime context by threads. Also the use of any client calls within an active service method requires a new context.

The soapcpp2 code generator tool also generates a service proxy class for C++ client applications (and service objects for server applications) with the -i (or -j) option:

```
> soapcpp2 -i calc.h
```

The proxy is defined in:

soapcalcProxy.h	client proxy class
soapcalcProxy.cpp	client proxy class

Note: without the -i option only old-style service proxies and objects are generated, which are less flexible and no longer recommended. Use -j as an alternative to -i to generate classes with the same functionality, but that are not inherited from **struct** soap and use a pointer to a **struct** soap engine context that can be shared with other proxy and service class instances. This choice is also important when services are chained, see Section 7.2.8.

The generated C++ proxy class initializes the gSOAP runtime context and offers the service interface as a collection of methods:

```
#include "soapcalcProxy.h" // get proxy
#include "calc.nsmap" // import the generated namespace mapping table
int main()
{
  calcProxy calc(SOAP_XML_INDENT);
  double sum;
  if (calc.add(1.0, 2.0, sum) == SOAP_OK)
    std::cout << "Sum = " << sum << std::endl;
  else
    calc.soap_stream_fault(std::cerr);
  return calc.error; // nonzero when error
}
```

The proxy class is derived from the gSOAP runtime context structure **struct** soap and thus inherits (option -i) all state information of the runtime. The proxy constructor takes context mode parameters to initialize the context, e.g. SOAP_XML_INDENT in this example.

The code is compiled and linked with soapcalcProxy.cpp, soapC.cpp, and stdsoap2.cpp (or use libgsoap++.a).

The proxy class name is extracted from the WSDL content and may not always be in a short format. Feel free to change the entry

```
//gsoap ns2 service name: calc
```

and rerun soapcpp2 to generate code that uses the new name.

When the example client application is invoked, a SOAP request is performed:

```
POST / engelen/calcserver.cgi HTTP/1.1
Host:   websrv.cs.fsu.edu
User-Agent:   gSOAP/2.7
Content-Type:   text/xml; charset=utf-8
Content-Length:   464
Connection:   close
```

```
    SOAPAction:   ""

    <?xml version="1.0" encoding="UTF-8"?>
    <SOAP-ENV:Envelope
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:c="urn:calc">
      <SOAP-ENV:Body SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <c:add>
          <a>1</a>
          <b>2</b>
        </c:add>
      </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
```

The SOAP response message:

```
    HTTP/1.1 200 OK
    Date:  Wed, 05 May 2010 16:02:21 GMT
    Server:  Apache/2.0.52 (Scientific Linux)
    Content-Length:  463
    Connection:  close
    Content-Type:  text/xml; charset=utf-8

    <?xml version="1.0" encoding="UTF-8"?>
    <SOAP-ENV:Envelope
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:ns="urn:calc">
      <SOAP-ENV:Body SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <ns:addResponse>
          <result>3</result>
        </ns:addResponse>
      </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
```

A client can invoke a sequence of service operations:

```
    #include "soapcalcProxy.h" // get proxy
    #include "calc.nsmap" // import the generated namespace mapping table
    int main()
    {
      calcProxy calc(SOAP_IO_KEEPALIVE); // keep-alive improves connection performance
      double sum = 0.0;
      double val[] = 5.0, 3.5, 7.1, 1.2 ;
      for (int i = 0; i < 4; i++)
        if (calc.add(sum, val[i], sum))
```

```
        return calc.error;
    std::cout << "Sum = " << sum << std::endl;
    return 0;
}
```

In the above, no data is deallocated until the proxy is deleted. To deallocate deserialized data between the calls, use:

```
for (int i = 0; i < 4; i++)
{
    if (calc.add(sum, val[i], sum))
        return calc.error;
    calc.destroy();
}
```

Deallocation is safe here, since the float values were copied and saved in sum. In other scenarios one must make sure data is copied or removed from the deallocation chain with:

```
soap_unlink(struct soap *soap, const void *data)
```

which is to be invoked on each data item to be preserved, before destroying deallocated data. When the proxy is deleted, also all deserialized data is deleted. To delegate deletion to another runtime context for later removal, use:

```
soap_delegate_deletion(struct soap *soap_from, struct soap *soap_to)
```

For example

```
struct soap soap;
soap_init(&soap);
{ // create proxy
    calcProxy calc;
    ... data generated ...
    soap_delegate_deletion(&calc, &soap);
} // proxy deleted
... data used ...
soap_destroy(&soap);
soap_end(&soap);
soap_done(&soap);
```

In C (use wsdl2h -c) the example program would be written as:

```
#include "soapH.h"
#include "calc.nsmap"
int main()
{
    struct soap soap;
    double sum = 0.0;
    double val[] =  5.0, 3.5, 7.1, 1.2 ;
```

```
      int i;
      for (i = 0; i < 4; i++)
      soap_init1(&soap, SOAP_IO_KEEPALIVE);
        if (soap_call_c__add(&soap, NULL, NULL, sum, val[i], &sum))
          return soap.error;
      printf("Sum = %lg\n", sum);
      soap_end(&soap);
      soap_done(&soap);
      return 0;
    }
```

The code above is compiled and linked with soapClient.c, soapC.c, and stdsoap2.c (or use libgsoap.a).

7.1.2   XML Namespace Considerations

The declaration of the ns2__add function prototype (discussed in the previous section) uses the namespace prefix ns2__ of the service operation namespace, which is distinguished by a **pair of underscores** in the function name to separate the namespace prefix from the service operation name. The purpose of a namespace prefix is to associate a service operation name with a service in order to prevent naming conflicts, e.g. to distinguish identical service operation names used by different services.

Note that the XML response of the service example uses a **namespace prefix** that may be different (e.g. ns) as long as it bound to the same **namespace name** urn:calc through the xmlns:ns="urn:calc binding. The use of namespace prefixes and namespace names is also required to enable SOAP applications to validate the content of SOAP messages. The namespace name in the service response is verified by the stub routine by using the information supplied in a **namespace mapping table** that is required to be part of gSOAP client and service application codes. The table is accessed at run time to resolve namespace bindings, both by the generated stub's data structure serializer for encoding the client request and by the generated stub's data structure deserializer to decode and validate the service response. The namespace mapping table should **not** be part of the header file input to the gSOAP soapcpp2 tool. Service details including namespace bindings may be provided with gSOAP directives in a header file, see Section 19.2.

The namespace mapping table is:

```
    struct Namespace namespaces[] =
    {  // {"ns-prefix", "ns-name"}
      {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"}, // MUST be first
      {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"}, // MUST be second
      {"xsi", "http://www.w3.org/2001/XMLSchema-instance"}, // MUST be third
      {"xsd", "http://www.w3.org/2001/XMLSchema"}, // 2001 XML Schema
      {"ns2", "urn:calc"}, // given by the service description
      {NULL, NULL} // end of table
    };
```

The first four namespace entries in the table consist of the standard namespaces used by the SOAP 1.1 protocol. In fact, the namespace mapping table is explicitly declared to enable a programmer

to specify the SOAP encoding style and to allow the inclusion of namespace-prefix with namespace-name bindings to comply to the namespace requirements of a specific SOAP service. For example, the namespace prefix `ns2`, which is bound to `urn:calc` by the namespace mapping table shown above, is used by the generated stub routine to encode the `add` request. This is performed automatically by the gSOAP `soapcpp2` tool by using the `ns2` prefix of the `ns2__add` method name specified in the `calc.h` header file. In general, if a function name of a service operation, **struct** name, **class** name, **enum** name, or field name of a **struct** or **class** has a pair of underscores, the name has a namespace prefix that must be defined in the namespace mapping table.

The namespace mapping table will be output as part of the SOAP Envelope by the stub routine. For example:

```
...
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns2="urn:calc"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
...
```

The namespace bindings will be used by a SOAP service to validate the SOAP request.

7.1.3   Example

The incorporation of namespace prefixes into C++ identifier names is necessary to distinguish service operations that share the same name but are provided by separate Web services and/or organizations. It avoids potential name clashes, while sticking to the C syntax. The C++ proxy classes generated with `soapcpp2 -i` (or -j) drop the namespace prefix from the method names

The namespace prefix convention is also be applied to non-primitive types. For example, **class** names are prefixed to avoid name clashes when the same name is used by multiple XML schemas. This ensures that the XML databinding never suffers from conflicting schema content. For example:

```
class e__Address // an electronic address from schema 'e'
{
  char *email;
  char *url;
};
class s__Address // a street address from schema 's'
{
  char *street;
  int number;
  char *city;
};
```

The namespace prefix is separated from the name of a data type by a pair of underscores (__).

An instance of `e__Address` is encoded by the generated serializer for this type as an Address element with namespace prefix `e`:

```
<e:Address xsi:type="e:Address">
  <email xsi:type="string">me@home</email>
  <url xsi:type="string">www.me.com</url>
</e:Address>
```

While an instance of s_Address is encoded by the generated serializer for this type as an Address element with namespace prefix s:

```
<s:Address xsi:type="s:Address">
  <street xsi:type="string">Technology Drive</street>
  <number xsi:type="int">5</number>
  <city xsi:type="string">Softcity</city>
</s:Address>
```

The namespace mapping table of the client program must have entries for e and s that refer to the XML Schemas of the data types:

```
struct Namespace namespaces[] =
{ ...
   {"e", "http://www.me.com/schemas/electronic-address"},
   {"s", "http://www.me.com/schemas/street-address"},
...
```

This table is required to be part of the client application to allow access by the serializers and deserializers of the data types at run time.

7.1.4   How to Generate C++ Client Proxy Classes

Proxy classes for C++ client applications are automatically generated by the gSOAP soapcpp2 tool, as was shown in Section 7.1.1.

There is a new and improved code generation capability for proxy classes, which is activated with the soapcpp2 -i (or j) option. These new proxy classes are derived from the soap structure, have a cleaner interface and offer more capabilites.

With C++, you can also use wsdl2h option -q*name* to generate the proxy in a C++ namespace *name*. This is very useful if you want to create multiple proxies for services by repeated use of wsdl2h and combine them in one code. Alternatively, you can run wsdl2h just once on all service WSDLs and have soapcpp2 generate multiple proxies for you. The latter approach does not use C++ namespaces and may reduce the overall amount of code.

To illustrate the generation of a "standard" (old-style) proxy class, the calc.h header file example of the previous section is augmented with the appropriate directives to enable the gSOAP soapcpp2 tool to generate the proxy class. Directives are included in the generated header file by the wsdl2h WSDL importer:

```
// Content of file "calc.h":
//gsoap ns2 service name: calc
//gsoap ns2 service port: http://websrv.cs.fsu.edu/ engelen/calcserver.cgi
```

```
//gsoap ns2 service protocol: SOAP1.1
//gsoap ns2 service style: rpc
//gsoap ns2 service encoding: encoded
//gsoap ns2 service namespace: urn:calc

//gsoap ns2 service method-protocol: add SOAP
//gsoap ns2 service method-style: add rpc
//gsoap ns2 service method-encoding: add encoded
//gsoap ns2 service method-action: add ""
int ns2__add(double a, double b, double& result);

//gsoap ns2 service method-protocol: sub SOAP
//gsoap ns2 service method-style: sub rpc
//gsoap ns2 service method-encoding: sub encoded
//gsoap ns2 service method-action: sub ""
int ns2__sub(double a, double b, double& result);

//gsoap ns2 service method-protocol: mul SOAP
//gsoap ns2 service method-style: mul rpc
//gsoap ns2 service method-encoding: mul encoded
//gsoap ns2 service method-action: mul ""
int ns2__mul(double a, double b, double& result);

...
```

The first three directives provide the service details, which is used to name the proxy class, the service location port (endpoint), and the XML namespace. The subsequent groups of three directives per method define the operation's SOAP style (RPC) and encoding (SOAP encoded), and SOAP action string. These directives can be provided for each service operation when the SOAPAction is required, such as with SOAP1.1 RPC encoded and when WS-Addressing is used. In this example, the service protocol is set by default for all operations to use SOAP 1.1 RPC encoding. For //gsoap directive details, see Section 19.2.

The soapcpp2 tool takes this header file and generates a proxy soapcalcProxy.h with the following contents (not using option -i):

```
#include "soapH.h"
class calc
{ public:
  struct soap *soap;
  const char *endpoint;
  calc() { ... };
  ~calc() { ... };
  virtual int ns2__add(double a, double b, double& result) { return soap ? soap_call_ns2__add(soap,
endpoint, NULL, a, b, result) : SOAP_EOM; };
  virtual int ns2__sub(double a, double b, double& result) { return soap ? soap_call_ns2__sub(soap,
endpoint, NULL, a, b, result) : SOAP_EOM; };
  virtual int ns2__mul(double a, double b, double& result) { return soap ? soap_call_ns2__mul(soap,
endpoint, NULL, a, b, result) : SOAP_EOM; };
  . . .
};
```

The gSOAP context and endpoint are declared public to enable access.

This generated proxy class can be included into a client application together with the generated namespace table as shown in this example:

```
#include "soapcalcProxy.h" // get proxy
#include "calc.nsmap" // get namespace bindings
int main()
{
    calc s;
    double r;
    if (s.ns2__add(1.0, 2.0, r) == SOAP_OK)
        std::cout << r << std::endl;
    else
        soap_print_fault(s.soap, stderr);
    return 0;
}
```

The constructor allocates and initializes a gSOAP context for the instance.

You can use soapcpp2 option -n together with -p to create a local namespaces table to avoid link conflicts when you need multiple namespace tables or need to combine multiple clients, see also Sections 9.1 and 19.36, and you can use a C++ code **namespace** to create a namespace qualified proxy class, see Section 19.35.

The soapcpp2 -i option to generate proxy classes derived from the base soap structure. In addition, these classes offer more functionality as illustrated in Section 7.1.1.

7.1.5   XSD Type Encoding Considerations

Many SOAP services require the explicit use of XML Schema types in the SOAP payload. The default encoding, which is also adopted by the gSOAP soapcpp2 tool, assumes SOAP RPC encoding which only requires the use of types to handle polymorphic cases. Nevertheless, the use of XSD typed messages is advised to improve interoperability. XSD types are introduced with **typedef** definitions in the header file input to the gSOAP soapcpp2 tool. The type name defined by a **typedef** definition corresponds to an XML Schema type (XSD type). For example, the following **typedef** declarations define various built-in XSD types implemented as primitive C/C++ types:

```
// Contents of header file:
...
typedef char *xsd__string; // encode xsd__string value as the xsd:string schema type
typedef char *xsd__anyURI; // encode xsd__anyURI value as the xsd:anyURI schema type
typedef float xsd__float; // encode xsd__float value as the xsd:float schema type
typedef long xsd__int; // encode xsd__int value as the xsd:int schema type
typedef bool xsd__boolean; // encode xsd__boolean value as the xsd:boolean schema type
typedef unsigned long long xsd__positiveInteger; // encode xsd__positiveInteger value as the
xsd:positiveInteger schema type
...
```

This easy-to-use mechanism informs the gSOAP soapcpp2 tool to generate serializers and deserializers that explicitly encode and decode the primitive C++ types as built-in primitive XSD types

when the `typedef`ed type is used in the parameter signature of a service operation (or when used nested within structs, classes, and arrays). At the same time, the use of **typedef** does not force any recoding of a C++ client or Web service application as the internal C++ types used by the application are not required to be changed (but still have to be primitive C++ types, see Section 11.3.2 for alternative class implementations of primitive XSD types which allows for the marshalling of polymorphic primitive types).

7.1.6   Example

Reconsider the calculator example, now rewritten with explicit XSD types to illustrate the effect:

```
// Contents of file "calc.h":
typedef double xsd__double;
int ns2__add(xsd__string a, xsd__double b, xsd__double &Result);
```

When processed by the gSOAP soapcpp2 tool it generates source code for the function soap_call_ns2__add, which is identical to the C-style SOAP call:

```
int soap_call_ns2__add(struct soap *soap, char *URL, char *action, double a, double b, double&
result);
```

The client application does not need to be rewritten and can still call the proxy using the "old" C-style function signatures. In contrast to the previous implementation of the stub however, the encoding and decoding of the data types by the stub has been changed to explicitly use the XSD types in the message payload.

For example, when the client application calls the proxy, the proxy produces a SOAP request with an `xsd:double` (the `xsi:type` is shown when the soapcpp2 `-t` option is used):

```
...
<SOAP-ENV:Body>
  <ns2:add>
    <a xsi:type="xsd:string">1.0</a>
    <b xsi:type="xsd:string">2.0</b>
  </ns2:add>
</SOAP-ENV:Body>
...
```

The service response is:

```
...
<soap:Body>
  <n:addResponse xmlns:n="urn:calc">
    <result xsi:type="xsd:double">3.0</result>
  </n:addResponse>
</soap:Body>
...
```

The validation of this service response by the stub routine takes place by matching the namespace names (URIs) that are bound to the `xsd` namespace prefix. The stub also expects the `addResponse` element to be associated with URI `urn:calc` through the binding of the namespace prefix `ns2` in the namespace mapping table. The service response uses namespace prefix `n` for the `addResponse` element. This namespace prefix is bound to the same URI `urn:calc` and therefore the service response is valid. When the XML is not well formed or does not pass validation, the response is rejected and a SOAP fault is generated. The validation level can be increased with the SOAP_XML_STRICT flag, but this is not advised for SOAP RPC encoded messaging.

7.1.7   How to Change the Response Element Name

There is no standardized convention for the response element name in a SOAP RPC encoded response message, although it is recommended that the response element name is the method name ending with "`Response`". For example, the response element of `add` is `addResponse`.

The response element name can be specified explicitly using a **struct** or **class** declaration in the header file. This name must be qualified by a namespace prefix, just as the operation name should use a namespace prefix. The **struct** or **class** name represents the SOAP response element name used by the service. Consequently, the output parameter of the service operation must be declared as a field of the **struct** or **class**. The use of a **struct** or a **class** for the service response is fully SOAP 1.1 compliant. In fact, the absence of a **struct** or **class** indicates to the soapcpp2 tool to automatically generate a struct for the response which is internally used by a stub.

7.1.8   Example

Reconsider the calculator service operation specification which can be rewritten with an explicit declaration of a SOAP response element as follows:

```
// Contents of "calc.h":
typedef double xsd__double;
struct ns2__addResponse {xsd__double result;};
int ns2__add(xsd__string a, xsd__double b, struct ns2__addResponse &r);
```

The SOAP request and response messages are the same as before:

```
...
<SOAP-ENV:Body>
  <ns2:add>
    <a xsi:type="xsd:string">1.0</a>
    <b xsi:type="xsd:string">2.0</b>
  </ns2:add>
</SOAP-ENV:Body>
...
```

The difference is that the service response is required to match the specified `addResponse` name and its namespace URI:

```
...
<soap:Body>
  <n:addResponse xmlns:n='urn:calc'>
    <result xsi:type="xsd:double">3.0</result>
  </n:addResponse>
</soap:Body>
...
```

This use of a **struct** or **class** enables the adaptation of the default SOAP response element name and/or namespace URI when required.

7.1.9   How to Specify Multiple Output Parameters

The gSOAP soapcpp2 tool compiler uses the convention that the **last parameter** of the function prototype declaration of a service operation in a header file is also the **only single output parameter** of the method. All other parameters are considered input parameters of the service operation. To specify a service operation with **multiple output parameters**, a **struct** or **class** must be declared for the service operation response, see also 7.1.7. The name of the **struct** or **class** must have a namespace prefix, just as the service method name. The fields of the **struct** or **class** are the output parameters of the service operation. Both the order of the input parameters in the function prototype and the order of the output parameters (the fields in the **struct** or **class**) is not significant. However, the SOAP 1.1 specification states that input and output parameters may be treated as having anonymous parameter names which requires a particular ordering, see Section 7.1.13.

7.1.10   Example

As an example, consider a hypothetical service operation getNames with a single input parameter SSN and two output parameters first and last. This can be specified as:

```
// Contents of file "getNames.h":
int ns3__getNames(char *SSN, struct ns3__getNamesResponse {char *first; char *last;} &r);
```

The gSOAP soapcpp2 tool takes this header file as input and generates source code for the function soap_call_ns3__getNames. When invoked by a client application, the proxy produces the SOAP request:

```
...
<SOAP-ENV:Envelope ...  xmlns:ns3="urn:names" ...>
...
<ns3:getNames>
<SSN>999 99 9999</SSN>
</ns3:getNames>
...
```

The response by a SOAP service could be:

```
...
<m:getNamesResponse xmlns:m="urn:names">
```

```
<first>John</first>
<last>Doe</last>
</m:getNamesResponse>
...
```

where `first` and `last` are the output parameters of the `getNames` service operation of the service.

As another example, consider a service operation `copy` with an input parameter and an output parameter with identical parameter names (this is not prohibited by the SOAP 1.1 protocol). This can be specified as well using a response **struct**:

```
// Content of file "copy.h":
int X_rox_ _copy_name(char *name, struct X_rox_ _copy_nameResponse {char *name;} &r);
```

The use of a **struct** or **class** for the service operation response enables the declaration of service operations that have parameters that are passed both as input and output parameters.

The gSOAP soapcpp2 compiler takes the copy.h header file as input and generates the soap_call_X_rox_ _copy_name proxy. When invoked by a client application, the proxy produces the SOAP request:

```
...
<SOAP-ENV:Envelope ...  xmlns:X-rox="urn:copy" ...>
...
<X-rox:copy-name>
<name>SOAP</name>
</X-rox:copy-name>
...
```

The response by a SOAP copy service could be something like:

```
...
<m:copy-nameResponse xmlns:m="urn:copy">
<name>SOAP</name>
</m:copy-nameResponse>
...
```

The name will be parsed and decoded by the proxy and returned in the `name` field of the **struct** X_rox_ _copy_nameResponse &r parameter.

7.1.11   How to Specify Output Parameters With struct/class Compound Data Types

If the single output parameter of a service operation is a complex data type such as a **struct** or **class** it is necessary to specify the response element of the service operation as a **struct** or **class at all times**. Otherwise, the output parameter will be considered the response element (!), because of the response element specification convention used by gSOAP, as discussed in 7.1.7.

7.1.12   Example

This is best illustrated with an example. The Flighttracker service by ObjectSpace provides real time flight information for flights in the air. It requires an airline code and flight number as

parameters. The service operation name is getFlightInfo and the method has two string parameters: the airline code and flight number, both of which must be encoded as `xsd:string` types. The method returns a getFlightResponse response element with a return output parameter that is of complex type FlightInfo. The type FlightInfo is represented by a **class** in the header file, whose field names correspond to the FlightInfo accessors:

```
// Contents of file "flight.h":
typedef char *xsd__string;
class ns2__FlightInfo
{
  public:
  xsd__string airline;
  xsd__string flightNumber;
  xsd__string altitude;
  xsd__string currentLocation;
  xsd__string equipment;
  xsd__string speed;
};
struct ns1__getFlightInfoResponse {ns2__FlightInfo return_;};
int ns1__getFlightInfo(xsd__string param1, xsd__string param2, struct ns1__getFlightInfoResponse
&r);
```

The response element ns1__getFlightInfoResponse is explicitly declared and it has one field: `return_` of type ns2__FlightInfo. Note that `return_` has a trailing underscore to avoid a name clash with the **return** keyword, see Section 10.3 for details on the translation of C++ identifiers to XML element names.

The gSOAP soapcpp2 compiler generates the soap_call_ns1__getFlightInfo proxy. Here is an example fragment of a client application that uses this proxy to request flight information:

```
struct soap soap;
...
soap_init(&soap);
...
soap_call_ns1__getFlightInfo(&soap, "testvger.objectspace.com/soap/servlet/rpcrouter",
  "urn:galdemo:flighttracker", "UAL", "184", r);
...
struct Namespace namespaces[] =
{
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
  {"SOAP-ENC","http://schemas.xmlsoap.org/soap/encoding/"},
  {"xsi", "http://www.w3.org/2001/XMLSchema-instance"},
  {"xsd", "http://www.w3.org/2001/XMLSchema"},
  {"ns1", "urn:galdemo:flighttracker"},
  {"ns2", "http://galdemo.flighttracker.com"},
  {NULL, NULL}
};
```

When invoked by a client application, the proxy produces the SOAP request:

```
POST /soap/servlet/rpcrouter HTTP/1.1
Host:  testvger.objectspace.com
```

```
Content-Type:  text/xml
Content-Length:  634
SOAPAction:  "urn:galdemo:flighttracker"

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns1="urn:galdemo:flighttracker"
  xmlns:ns2="http://galdemo.flighttracker.com"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<ns1:getFlightInfo xsi:type="ns1:getFlightInfo">
<param1 xsi:type="xsd:string">UAL</param1>
<param2 xsi:type="xsd:string">184</param2>
</ns1:getFlightInfo>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The Flighttracker service responds with:

```
HTTP/1.1 200 ok
Date:  Thu, 30 Aug 2001 00:34:17 GMT
Server:  IBM_HTTP_Server/1.3.12.3 Apache/1.3.12 (Win32)
Set-Cookie:  sesessionid=2GFVTOGC30D0LGRGU2L4HFA;Path=/
Cache-Control:  no-cache="set-cookie,set-cookie2"
Expires:  Thu, 01 Dec 1994 16:00:00 GMT
Content-Length:  861
Content-Type:  text/xml; charset=utf-8
Content-Language:  en

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getFlightInfoResponse xmlns:ns1="urn:galdemo:flighttracker"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xmlns:ns2="http://galdemo.flighttracker.com" xsi:type="ns2:FlightInfo">
<equipment xsi:type="xsd:string">A320</equipment>
<airline xsi:type="xsd:string">UAL</airline>
<currentLocation xsi:type="xsd:string">188 mi W of Lincoln, NE</currentLocation>
<altitude xsi:type="xsd:string">37000</altitude>
<speed xsi:type="xsd:string">497</speed>
<flightNumber xsi:type="xsd:string">184</flightNumber>
</return>
</ns1:getFlightInfoResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The proxy returns the service response in variable r of type **struct** ns1_ _getFlightInfoResponse and this information can be displayed by the client application with the following code fragment:

```
cout << r.return_.equipment << " flight " << r.return_.airline << r.return_.flightNumber
   << " traveling " << r.return_.speed << " mph " << " at " << r.return_.altitude
   << " ft, is located " << r.return_.currentLocation << endl;
```

This code displays the service response as:

```
A320 flight UAL184 traveling 497 mph at 37000 ft, is located 188 mi W of Lincoln,
NE
```

Note: the flight tracker service is no longer available since 9/11/2001. It is kept in the documentation as an example to illustrate the use of structs/classes and response types.

7.1.13   How to Specify Anonymous Parameter Names

The SOAP RPC encoding protocol allows parameter names to be anonymous. That is, the name(s) of the output parameters of a service operation are not strictly required to match a client's view of the parameters names. Also, the input parameter names of a service operation are not strictly required to match a service's view of the parameter names. The gSOAP soapcpp2 compiler can generate stub and skeleton routines that support anonymous parameters. Parameter names are implicitly anonymous by omitting the parameter names in the function prototype of the service operation. For example:

```
// Contents of "calc.h":
typedef double xsd_ _double;
int ns2_ _add(xsd_ _string, xsd_ _double, xsd_ _double &);
```

To make parameter names explicitly anonymous on the receiving side (client or service), the parameter names should start with an underscore (_) in the function prototype in the header file.

For example:

```
// Contents of "calc.h":
typedef double xsd_ _double;
int ns2_ _add(xsd_ _string _a, xsd_ _double _b, xsd_ _double & _return);
```

In this example, the _a, _b, and _return are anonymous parameters. As a consequence, the service response to a request made by a client created with gSOAP using this header file specification may include any name for the output parameter in the SOAP payload. The input parameters may also be anonymous. This affects the implementation of Web services in gSOAP and the matching of parameter names by the service.

Caution: when anonymous parameter names are used, the order of the parameters in the function prototype of a service operation is significant.

7.1.14 How to Specify a Method with No Input Parameters

To specify a service operation that has no input parameters, just provide a function prototype with one parameter which is the output parameter (some C/C++ compilers will not compile and complain about an empty **struct**: use compile flag -DWITH_NOEMPTYSTRUCT to compile the generated code for these cases). This **struct** is generated by gSOAP to contain the SOAP request message. To fix this, provide one input parameter of type **void*** (gSOAP can not serialize void* data). For example:

```
struct ns3__SOAPService
{
  public:
  int ID;
  char *name;
  char *owner;
  char *description;
  char *homepageURL;
  char *endpoint;
  char *SOAPAction;
  char *methodNamespaceURI;
  char *serviceStatus;
  char *methodName;
  char *dateCreated;
  char *downloadURL;
  char *wsdlURL;
  char *instructions;
  char *contactEmail;
  char *serverImplementation;
};
struct ArrayOfSOAPService {struct ns3__SOAPService *__ptr; int __size;};
int ns__getAllSOAPServices(void *_, struct ArrayOfSOAPService &_return);
```

The ns__getAllSOAPServices method has one **void*** input parameter which is ignored by the serializer to produce the request message.

Most C/C++ compilers allow empty **struct**s and therefore the **void*** parameter is not required.

7.1.15 How to Specify a Method with No Output Parameters

To specify a service operation that has no output parameters, just define a function prototype with a response struct that is empty. For example:

```
enum ns__event { off, on, stand_by };
int ns__signal(enum ns__event in, struct ns__signalResponse { } *out);
```

Since the response struct is empty, no output parameters are specified.

Some SOAP resources refer to SOAP RPC with empty responses as **one way** SOAP messaging. However, we refer to one-way massaging by asynchronous explicit send and receive operations as described in Section 7.3. The latter view of one-way SOAP messaging is also in line with Basic Profile 1.0.

7.2 How to Build SOAP/XML Web Services

The gSOAP soapcpp2 compiler generates **skeleton** routines in C++ source form for each of the service operations specified as function prototypes in the header file processed by the gSOAP soapcpp2 compiler. The skeleton routines can be readily used to implement the service operations in a new SOAP Web service. The compound data types used by the input and output parameters of service operations must be declared in the header file, such as structs, classes, arrays, and pointer-based data structures (graphs) that are used as the data types of the parameters of a service operation. The gSOAP soapcpp2 compiler automatically generates serializers and deserializers for the data types to enable the generated skeleton routines to encode and decode the contents of the parameters of the service operations. The gSOAP soapcpp2 compiler also generates a service operation request dispatcher routine that will serve requests by calling the appropriate skeleton when the SOAP service application is installed as a CGI application on a Web server.

7.2.1 Example

The following example specifies three service operations to be implemented by a new SOAP Web service:

```
// Contents of file "calc.h":
typedef double xsd__double;
int ns__add(xsd__double a, xsd__double b, xsd__double &result);
int ns__sub(xsd__double a, xsd__double b, xsd__double &result);
int ns__sqrt(xsd__double a, xsd__double &result);
```

The add and sub methods are intended to add and subtract two double floating point numbers stored in input parameters a and b and should return the result of the operation in the result output parameter. The sqrt method is intended to take the square root of input parameter a and to return the result in the output parameter result. The xsd__double type is recognized by the gSOAP soapcpp2 compiler as the `xsd:double` XSD Schema data type. The use of **typedef** is a convenient way to associate primitive C types with primitive XML Schema data types.

To generate the skeleton routines, the gSOAP soapcpp2 compiler is invoked from the command line with:

```
> soapcpp2 calc.h
```

The compiler generates the skeleton routines for the add, sub, and sqrt service operations specified in the calc.h header file. The skeleton routines are respectively, soap_serve_ns__add, soap_serve_ns__sub, and soap_serve_ns__sqrt and saved in the file soapServer.cpp. The generated file soapC.cpp contains serializers and deserializers for the skeleton. The compiler also generates a service dispatcher: the soap_serve function handles client requests on the standard input stream and dispatches the service operation requests to the appropriate skeletons to serve the requests. The skeleton in turn calls the service operation implementation function. The function prototype of the service operation implementation function is specified in the header file that is input to the gSOAP soapcpp2 compiler.

Here is an example Calculator service application that uses the generated soap_serve routine to handle client requests:

44

```cpp
// Contents of file "calc.cpp":
#include "soapH.h"
#include <math.h> // for sqrt()
int main()
{
   return soap_serve(soap_new()); // use the service operation request dispatcher
}
// Implementation of the "add" service operation:
int ns__add(struct soap *soap, double a, double b, double &result)
{
   result = a + b;
   return SOAP_OK;
}
// Implementation of the "sub" service operation:
int ns__sub(struct soap *soap, double a, double b, double &result)
{
   result = a - b;
   return SOAP_OK;
}
// Implementation of the "sqrt" service operation:
int ns__sqrt(struct soap *soap, double a, double &result)
{
   if (a >= 0)
   {
      result = sqrt(a);
      return SOAP_OK;
   }
   else
      return soap_receiver_fault(soap, "Square root of negative number", "I can only take the square
root of a non-negative number");
}
// As always, a namespace mapping table is needed:
struct Namespace namespaces[] =
{   // {"ns-prefix", "ns-name"}
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
   {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
   {"xsi", "http://www.w3.org/2001/XMLSchema-instance"},
   {"xsd", "http://www.w3.org/2001/XMLSchema"},
   {"ns", "urn:simple-calc"}, // bind "ns" namespace prefix
   {NULL, NULL}
};
```

Note that the service operations have an extra input parameter which is a pointer to the gSOAP runtime context. The implementation of the service operations MUST return a SOAP error code. The code SOAP_OK denotes success, while SOAP_FAULT denotes an exception with details that can be defined by the user. The exception description can be assigned to the soap->fault->faultstring string and details can be assigned to the soap->fault->detail string. This is SOAP 1.1 specific. SOAP 1.2 requires the soap->fault->SOAP_ENV__Reason and the soap->fault->SOAP_ENV__Detail strings to be assigned. Better is to use the soap_receiver_fault function that allocates a fault struct and sets the SOAP Fault string and details regardless of the SOAP 1.1 or SOAP 1.2 version used. The

soap_receiver_fault function returns SOAP_FAULT, i.e. an application-specific fault. The fault exception will be passed on to the client of this service.

This service application can be readily installed as a CGI application. The service description would be:

Endpoint URL:	the URL of the CGI application
SOAP action:	"" (2 quotes)
Remote method namespace:	`urn:simple-calc`
Remote method name:	`add`
Input parameters:	`a` of type `xsd:double` and `b` of type `xsd:double`
Output parameter:	`result` of type `xsd:double`
Remote method name:	`sub`
Input parameters:	`a` of type `xsd:double` and `b` of type `xsd:double`
Output parameter:	`result` of type `xsd:double`
Remote method name:	`sqrt`
Input parameter:	`a` of type `xsd:double`
Output parameter:	`result` of type `xsd:double` or a SOAP Fault

The soapcpp2 compile generates a WSDL file for this service, see Section 7.2.9.

Unless the CGI application inspects and checks the environment variable SOAPAction which contains the SOAP action request by a client, the SOAP action is ignored by the CGI application. SOAP actions are specific to the SOAP protocol and provide a means for routing requests and for security reasons (e.g. firewall software can inspect SOAP action headers to grant or deny the SOAP request. Note that this requires the SOAP service to check the SOAP action header as well to match it with the service operation.)

The header file input to the gSOAP soapcpp2 compiler does not need to be modified to generate client stubs for accessing this service. Client applications can be developed by using the same header file as for which the service application was developed. For example, the soap_call_ns__add stub routine is available from the soapClient.cpp file after invoking the gSOAP soapcpp2 compiler on the calc.h header file. As a result, client and service applications can be developed without the need to know the details of the SOAP encoding used.

7.2.2 MSVC++ Builds

- Win32 builds need winsock2 (MS Visual C++ "ws2_32.lib") To do this in Visual C++ 6.0, go to "Project", "settings", select the "Link" tab (the project file needs to be selected in the file view) and add "ws2_32.lib" to the "Object/library modules" entry.

- Use files with extension .cpp only (don't mix .c with .cpp).

- Turn pre-compiled headers off.

- When creating a new project, you can specify a custom build step to automatically invoke the gSOAP soapcpp2 compiler on a gSOAP header file. In this way you can incrementally build a new service by adding new operations and data types to the header file. To specify a custom build step, select the "Project" menu item "Settings" and select the header file in the File view pane. Select the "Custom Build" tab and enter 'soapcpp2.exe "$(inputPath)"' in

46

the "Command" pane. Enter 'soapStub.h soapH.h soapC.cpp soapClient.cpp soapServer.cpp'. Don't forget to add the soapXYZProxy.h soapXYZObject.h files that are generated for C++ class proxies and server objects named XYZ. Click "OK". Run soapcpp2 once to generate these files (you can simply do this by selecting your header file and select "Compile"). Add the files to your project. Each time you make a change to the header file, the project sources are updated automatically.

- You may want to use the WinInet interface available in the mod_gsoap directory of the gSOAP package to simplify Internet access and deal with encryption, proxies, and authentication. API instructions are included in the source.

- For the PocketPC, run the wsdl2h WSDL parser with option -s to prevent the generation of STL code. In addition, time_t serialization is not supported, which means that you should add the following line to typemap.dat indicating a mapping of xsd__dateTime to **char***: xsd__dateTime = | char* | char*.

7.2.3 How to Create a Stand-Alone Server

The deployment of a Web service as a CGI application is an easy means to provide your service on the Internet. However, the performance of CGI is not great. Also, gSOAP services can be run as stand-alone services on any port by utilizing the built-in HTTP and TCP/IP stacks. However, the preferred mechanism to deploy a service is through an Apache module or IIS module. These servers and modules are designed for server load balancing and access control.

To create a stand-alone service, only the main routine of the service needs to be modified as follows. Instead of just calling the soap_serve routine, the main routine is changed into:

```
int main()
{
  struct soap soap;
  int m, s; // master and slave sockets
  soap_init(&soap);
  m = soap_bind(&soap, "machine.genivia.com", 18083, 100);
  if (m < 0)
    soap_print_fault(&soap, stderr);
  else
  {
    fprintf(stderr, "Socket connection successful: master socket = %d\n", m);
    for (int i = 1; ; i++)
    {
      s = soap_accept(&soap);
      if (s < 0)
      {
        soap_print_fault(&soap, stderr);
        break;
      }
      fprintf(stderr, "%d: accepted connection from IP=%d.%d.%d.%d socket=%d", i,
        (soap.ip>>24)&0xFF, (soap.ip>>16)&0xFF, (soap.ip>>8)&0xFF, soap.ip&0xFF, s);
      if (soap_serve(&soap) != SOAP_OK) // process RPC request
```

```
          soap_print_fault(&soap, stderr); // print error
        fprintf(stderr, "request served\n");
        soap_destroy(&soap); // clean up class instances
        soap_end(&soap); // clean up everything and close socket
      }
    }
    soap_done(&soap); // close master socket and detach context
  }
```

The soap_serve dispatcher handles one request or multiple requests when HTTP keep-alive is enabled
(with the SOAP_IO_KEEPALIVE flag see Section 19.11).

The gSOAP functions that are frequently used for server-side coding are:

Function	Description
soap_new()	Allocates and Initializes gSOAP context
soap_init(**struct** soap *soap)	Initializes a stack-allocated gSOAP context (required once)
soap_bind(**struct** soap *soap, **char** *host, **int** port, **int** backlog)	Returns master socket (backlog = max. queue size for requests). When host==NULL: host is the machine on which the service runs
soap_accept(**struct** soap *soap)	Returns slave socket
soap_end(**struct** soap *soap)	Clean up deserialized data (except class instances) and temporary data
soap_free_temp(**struct** soap *soap)	Clean up temporary data only
soap_destroy(**struct** soap *soap)	Clean up deserialized class instances (note: this function will be renamed with option -n
soap_done(**struct** soap *soap)	Reset and detach context: close master/slave sockets and remove callbacks
soap_free(**struct** soap *soap)	Detach and deallocate context (soap_new())

The *host* name in soap_bind may be NULL to indicate that the current host should be used.

The soap.accept_timeout attribute of the gSOAP runtime context specifies the timeout value for a
non-blocking soap_accept(&soap) call. See Section 19.19 for more details on timeout management.

See Section 9.13 for more details on memory management.

A client application connects to this stand-alone service with the endpoint machine.genivia.com:18083.
A client may use the http:// prefix. When absent, no HTTP header is sent and no HTTP-based
information will be communicated to the service.

7.2.4   How to Create a Multi-Threaded Stand-Alone Service

Stand-alone multi-threading a Web Service is essential when the response times for handling requests by the service are (potentially) long or when keep-alive is enabled, see Section 19.11. In case
of long response times, the latencies introduced by the unrelated requests may become prohibitive
for a successful deployment of a stand-alone service. When HTTP keep-alive is enabled, a client
may not close the socket on time, thereby preventing other clients from connecting.

However, the preferred mechanism to deploy a service is through an Apache module or IIS module.
These servers and modules are designed for server load balancing and access control.

The following example illustrates the use of threads to improve the quality of service by handling new requests in separate threads:

```
#include "soapH.h"
#include <pthread.h>
#define BACKLOG (100) // Max. request backlog
int main(int argc, char **argv)
{
  struct soap soap;
  soap_init(&soap);
  if (argc < 2) // no args: assume this is a CGI application
  {
    soap_serve(&soap); // serve request, one thread, CGI style
    soap_destroy(&soap); // dealloc C++ data
    soap_end(&soap); // dealloc data and clean up
  }
  else
  {
    soap.send_timeout = 60; // 60 seconds
    soap.recv_timeout = 60; // 60 seconds
    soap.accept_timeout = 3600; // server stops after 1 hour of inactivity
    soap.max_keep_alive = 100; // max keep-alive sequence
    void *process_request(void*);
    struct soap *tsoap;
    pthread_t tid;
    int port = atoi(argv[1]); // first command-line arg is port
    SOAP_SOCKET m, s;
    m = soap_bind(&soap, NULL, port, BACKLOG);
    if (!soap_valid_socket(m))
      exit(1);
    fprintf(stderr, "Socket connection successful %d\n", m);
    for (;;)
    {
      s = soap_accept(&soap);
      if (!soap_valid_socket(s))
      {
        if (soap.errnum)
        {
          soap_print_fault(&soap, stderr);
          exit(1);
        }
        fprintf(stderr, "server timed out\n");
        break;
      }
      fprintf(stderr, "Thread %d accepts socket %d connection from IP %d.%d.%d.%d\n",
i, s, (soap.ip>>24)&0xFF, (soap.ip>>16)&0xFF, (soap.ip>>8)&0xFF, soap.ip&0xFF);
      tsoap = soap_copy(&soap); // make a safe copy
      if (!tsoap)
        break;
      pthread_create(&tid, NULL, (void*(*)(void*))process_request, (void*)tsoap);
    }
```

```
    }
    soap_done(&soap); // detach soap struct
    return 0;
}
void *process_request(void *soap)
{
    pthread_detach(pthread_self());
    soap_serve((struct soap*)soap);
    soap_destroy((struct soap*)soap); // dealloc C++ data
    soap_end((struct soap*)soap); // dealloc data and clean up
    soap_done((struct soap*)soap); // detach soap struct
    free(soap);
    return NULL;
}
```

Note: the code does not wait for threads to join the main thread upon program termination.

The soap_serve dispatcher handles one request or multiple requests when HTTP keep-alive is set with SOAP_IO_KEEPALIVE. The soap.max_keep_alive value can be set to the maximum keep-alive calls allowed, which is important to avoid a client from holding a thread indefinitely. The send and receive timeouts are set to avoid (intentionally) slow clients from holding a socket connection too long. The accept timeout is used to let the server terminate automatically after a period of inactivity.

The following example uses a pool of servers to limit the machine's resource utilization:

```
#include "soapH.h"
#include <pthread.h>
#define BACKLOG (100) // Max. request backlog
#define MAX_THR (10) // Max. threads to serve requests
int main(int argc, char **argv)
{
    struct soap soap;
    soap_init(&soap);
    if (argc < 2) // no args: assume this is a CGI application
    {
        soap_serve(&soap); // serve request, one thread, CGI style
        soap_destroy(&soap); // dealloc C++ data
        soap_end(&soap); // dealloc data and clean up
    }
    else
    {
        struct soap *soap_thr[MAX_THR]; // each thread needs a runtime context
        pthread_t tid[MAX_THR];
        int port = atoi(argv[1]); // first command-line arg is port
        SOAP_SOCKET m, s;
        int i;
        m = soap_bind(&soap, NULL, port, BACKLOG);
        if (!soap_valid_socket(m))
            exit(1);
        fprintf(stderr, "Socket connection successful %d\n", m);
        for (i = 0; i < MAX_THR; i++)
            soap_thr[i] = NULL;
```

```
      for (;;)
      {
        for (i = 0; i < MAX_THR; i++)
        {
          s = soap_accept(&soap);
          if (!soap_valid_socket(s))
          {
            if (soap.errnum)
            {
              soap_print_fault(&soap, stderr);
              continue; // retry
            }
            else
            {
              fprintf(stderr, "Server timed out\n");
              break;
            }
          }
          fprintf(stderr, "Thread %d accepts socket %d connection from IP %d.%d.%d.%d\n",
i, s, (soap.ip>>24)&0xFF, (soap.ip>>16)&0xFF, (soap.ip>>8)&0xFF, soap.ip&0xFF);
          if (!soap_thr[i]) // first time around
          {
            soap_thr[i] = soap_copy(&soap);
            if (!soap_thr[i])
            exit(1); // could not allocate
          }
          else// recycle soap context
          {
            pthread_join(tid[i], NULL);
            fprintf(stderr, "Thread %d completed\n", i);
            soap_destroy(soap_thr[i]); // deallocate C++ data of old thread
            soap_end(soap_thr[i]); // deallocate data of old thread
          }
          soap_thr[i]->socket = s; // new socket fd
          pthread_create(&tid[i], NULL, (void*(*)(void*))soap_serve, (void*)soap_thr[i]);
        }
      }
      for (i = 0; i < MAX_THR; i++)
        if (soap_thr[i])
        {
          soap_done(soap_thr[i]); // detach context
          free(soap_thr[i]); // free up
        }
    }
    return 0;
  }
```

The following functions can be used to setup a gSOAP runtime context (**struct** soap):

Function	Description
soap_init(**struct** soap *soap)	Initializes a runtim
struct soap *soap_new()	Allocates, initialize
	context
struct soap *soap_copy(**struct** soap *soap)	Allocates a new run
	copy, i.e. the new c
	other context)
the argument context such that the new context does not share data with the argument context	
soap_done(**struct** soap *soap)	Reset, close commu

A new context is initiated for each thread to guarantee exclusive access to runtime contexts.

For clean termination of the server, the master socket can be closed and callbacks removed with soap_done(**struct** soap *soap).

The advantage of the code shown above is that the machine cannot be overloaded with requests, since the number of active services is limited. However, threads are still started and terminated. This overhead can be eliminated using a queue of requests (open sockets) as is shown in the code below.

```
#include "soapH.h"
#include <pthread.h>
#define BACKLOG (100) // Max. request backlog
#define MAX_THR (10) // Size of thread pool
#define MAX_QUEUE (1000) // Max. size of request queue
SOAP_SOCKET queue[MAX_QUEUE]; // The global request queue of sockets
int head = 0, tail = 0; // Queue head and tail
void *process_queue(void*);
int enqueue(SOAP_SOCKET);
SOAP_SOCKET dequeue();
pthread_mutex_t queue_cs;
pthread_cond_t queue_cv;
int main(int argc, char **argv)
{
  struct soap soap;
  soap_init(&soap);
  if (argc < 2) // no args: assume this is a CGI application
  {
    soap_serve(&soap); // serve request, one thread, CGI style
    soap_destroy(&soap); // dealloc C++ data
    soap_end(&soap); // dealloc data and clean up
  }
  else
  {
    struct soap *soap_thr[MAX_THR]; // each thread needs a runtime context
    pthread_t tid[MAX_THR];
    int port = atoi(argv[1]); // first command-line arg is port
    SOAP_SOCKET m, s;
    int i;
    m = soap_bind(&soap, NULL, port, BACKLOG);
    if (!soap_valid_socket(m))
```

```
        exit(1);
      fprintf(stderr, "Socket connection successful %d\n", m);
      pthread_mutex_init(&queue_cs, NULL);
      pthread_cond_init(&queue_cv, NULL);
      for (i = 0; i < MAX_THR; i++)
      {
        soap_thr[i] = soap_copy(&soap);
        fprintf(stderr, "Starting thread %d\n", i);
        pthread_create(&tid[i], NULL, (void*(*)(void*))process_queue, (void*)soap_thr[i]);
      }
      for (;;)
      {
        s = soap_accept(&soap);
        if (!soap_valid_socket(s))
        {
          if (soap.errnum)
          {
            soap_print_fault(&soap, stderr);
            continue; // retry
          }
          else
          {
            fprintf(stderr, "Server timed out\n");
            break;
          }
        }
        fprintf(stderr, "Thread %d accepts socket %d connection from IP %d.%d.%d.%d\n",
i, s, (soap.ip>>24)&0xFF, (soap.ip>>16)&0xFF, (soap.ip>>8)&0xFF, soap.ip&0xFF);
        while (enqueue(s) == SOAP_EOM)
          sleep(1);
      }
      for (i = 0; i < MAX_THR; i++)
      {
        while (enqueue(SOAP_INVALID_SOCKET) == SOAP_EOM)
          sleep(1);
      }
      for (i = 0; i < MAX_THR; i++)
      {
        fprintf(stderr, "Waiting for thread %d to terminate... ", i);
        pthread_join(tid[i], NULL);
        fprintf(stderr, "terminated\n");
        soap_done(soap_thr[i]);
        free(soap_thr[i]);
      }
      pthread_mutex_destroy(&queue_cs);
      pthread_cond_destroy(&queue_cv);
    }
    soap_done(&soap);
    return 0;
}
void *process_queue(void *soap)
```

```
{
  struct soap *tsoap = (struct soap*)soap;
  for (;;)
  {
    tsoap->socket = dequeue();
    if (!soap_valid_socket(tsoap->socket))
      break;
    soap_serve(tsoap);
    soap_destroy(tsoap);
    soap_end(tsoap);
    fprintf(stderr, "served\n");
  }
  return NULL;
}
int enqueue(SOAP_SOCKET sock)
{
  int status = SOAP_OK;
  int next;
  pthread_mutex_lock(&queue_cs);
  next = tail + 1;
  if (next >= MAX_QUEUE)
    next = 0;
  if (next == head)
    status = SOAP_EOM;
  else
  {
    queue[tail] = sock;
    tail = next;
    pthread_cond_signal(&queue_cv);
  }
  pthread_mutex_unlock(&queue_cs);
  return status;
}
SOAP_SOCKET dequeue()
{
  SOAP_SOCKET sock;
  pthread_mutex_lock(&queue_cs);
  while (head == tail)
    pthread_cond_wait(&queue_cv, &queue_cs);
  sock = queue[head++];
  if (head >= MAX_QUEUE)
    head = 0;
  pthread_mutex_unlock(&queue_cs);
  return sock;
}
```

Note: the plugin/threads.h and plugin/threads.c code can be used for a portable implementation. Instead of POSIX calls, use MUTEX_LOCK, MUTEX_UNLOCK, and COND_WAIT. These are wrappers for Win API calls or POSIX calls.

7.2.5   How to Pass Application Data to Service Methods

The **void** *soap.user field can be used to pass application data to service methods. This field should be set before the soap_serve() call. The service method can access this field to use the application-dependent data. The following example shows how a non-static database handle is initialized and passed to the service methods:

```
{ ...
    struct soap soap;
    database_handle_type database_handle;
    soap_init(&soap);    soap.user = (void*)database_handle;
    ...
    soap_serve(&soap); // call the service operation dispatcher to handle request
    ...
}
int ns__myMethod(struct soap *soap, ...)
{ ...
    fetch((database_handle_type*)soap->user);
// get data    ...
    return SOAP_OK;
}
```

Another way to pass application data around in a more organized way is accomplished with plugins, see Section 19.38.

7.2.6   Web Service Implementation Aspects

The same client header file specification issues apply to the specification and implementation of a SOAP Web service. Refer to

- 7.1.2 for namespace considerations.

- 7.1.5 for an explanation on how to change the encoding of the primitive types.

- 7.1.7 for a discussion on how the response element format can be controlled.

- 7.1.9 for details on how to pass multiple output parameters from a service operation.

- 7.1.11 for passing complex data types as output parameters.

- 7.1.13 for anonymizing the input and output parameter names.

7.2.7   How to Generate C++ Server Object Classes

Server object classes for C++ server applications are automatically generated by the gSOAP soapcpp2 compiler.

There are two modes for generating classes. Use soapcpp2 option -i (or -j) to generate improved class definitions where the class' member functions are the service methods.

The older examples (without the use of soapcpp2 option -i and -j) use a C-like approach with globally defined service methods, which is illustated here with a calculator example:

```
// Content of file "calc.h":
//gsoap ns service name: Calculator
//gsoap ns service protocol: SOAP
//gsoap ns service style: rpc
//gsoap ns service encoding: encoded
//gsoap ns service location: http://www.cs.fsu.edu/~engelen/calc.cgi
//gsoap ns schema namespace: urn:calc
//gsoap ns service method-action: add ""
int ns__add(double a, double b, double &result);
int ns__sub(double a, double b, double &result);
int ns__mul(double a, double b, double &result);
int ns__div(double a, double b, double &result);
```

The first three directives provide the service name which is used to name the service class, the service location (endpoint), and the schema. The fourth directive defines the optional SOAPAction for the method, which is a string associated with SOAP 1.1 operations. Compilation of this header file with soapcpp2 -i creates a new file soapCalculatorObject.h with the following contents:

```
#include "soapH.h"
class CalculatorObject : public soap
{ public:
    Calculator() { ... };
    ~Calculator() { ... };
    int serve() { return soap_serve(this); };
};
```

This generated server object class can be included into a server application together with the generated namespace table as shown in this example:

```
#include "soapCalculatorObject.h" // get server object
#include "Calculator.nsmap" // get namespace bindings
int main()
{
    CalculatorObject c;
    return c.serve(); // calls soap_serve to serve as CGI application (using stdin/out)
}
// C-style global functions implement server operations (soapcpp2 w/o option -i)
int ns__add(struct soap *soap, double a, double b, double &result)
{
    result = a + b;
    return SOAP_OK;
}
... sub(), mul(), and div() implementations ...
```

You can use soapcpp2 option -n together with -p to create a local namespace table to avoid link conflict when you need to combine multiple tables and/or multiple servers, see also Sections 9.1

and 19.36, and you can use a C++ code **namespace** to create a namespace qualified server object class, see Section 19.35.

The example above serves requests over stdin/out. Use the bind and accept calls to create a stand-alone server to service inbound requests over sockets, see also 7.2.3.

A better alternative is to use the soapcpp2 option -i. The C++ proxy and server objects are derived from the soap context struct, which simplifies the proxy invocation and service operation implementations.

Compilation of the above header file with the gSOAP compiler soapcpp2 option -i creates new files soapCalculatorService.h and soapCalculatorService.cpp (rather than the C-style soapServer.cpp).

This generated server object class can be included into a server application together with the generated namespace table as shown in this example:

```
#include "soapCalculatorService.h" // get server object
#include "Calculator.nsmap" // get namespace bindings
int main()
{
  soapCalculatorService c;
  return c.serve(); // calls soap_serve to serve as CGI application (using stdin/out)
}
// The 'add' service method (soapcpp2 w/ option -i)
int soapCalculatorService::add(double a, double b, double &result)
{
  result = a + b;
  return SOAP_OK;
}
... sub(), mul(), and div() implementations ...
```

Note that the service operation does not need a prefix (ns__) and there is no soap context struct passed to the service operation since the service object itself is the context (it is derived from the soap struct).

7.2.8 How to Chain C++ Server Classes to Accept Messages on the Same Port

When combining multiple services into one application, you can run wsdl2h on multiple WSDLs to generate the single all-inclusive service definitions header file. This header file is then processed with soapcpp2, for example to generate server class objects with option -i and -q to separate the service codes with C++ namespaces, see Section 19.35.

This works well, but the problem is that we end up with multiple classes, each for a collection of service operations the class is supposed to implement. But what if we need to provide one endpoint port for all services and operations? In this case invoking the server object's serve method is not sufficient, since only one service can accept requests while we want multiple services to listen to the same port.

The approach is to chain the service dispatchers, as shown below:

```
Abc::soapABCService abc; // generated with soapcpp2 -i -S -qAbc
Uvw::soapUVWService uvw; // generated with soapcpp2 -i -S -qUvw
```

```
Xyz::soapXYZService xyz; // generated with soapcpp2 -i -S -qXyz
...
abc.bind(NULL, 8080, 100);
...
abc.accept();
// when using SSL: ssl_accept(&abc);
...
if(soap_begin_serve(&abc)) // available in 2.8.2 and later
   abc.soap_stream_fault(std::cerr);
elseif (abc.dispatch() == SOAP_NO_METHOD)
{
   soap_copy_stream(&uvw, &abc);
   if (uvw.dispatch() == SOAP_NO_METHOD)
   {
      soap_copy_stream(&xyz, &uvw);
      if (xyz.dispatch())
      {
         soap_send_fault(&xyz); // send fault to client
         xyz.soap_stream_fault(std::cerr);
      }
      soap_free_stream(&xyz); // free the copy
      xyz.destroy();
   }
   else
   {
      soap_send_fault(&uvw); // send fault to client
      uvw.soap_stream_fault(std::cerr);
   }
   soap_free_stream(&uvw); // free the copy
   uvw.destroy();
}
else
   abc.soap_stream_fault(std::cerr);
abc.destroy();
...
```

The dispatch method parses the SOAP/XML request and invokes the service operations, unless there is no matching operation and SOAP_NO_METHOD is returned. The soap_copy_stream ensures that the service object uses the currently open socket. The copied streams are freed with soap_free_stream. Do not enable keep-alive support, as the socket may stay open indefinitely afterwards as a consequence. Also, the dispatch method does not send a fault to the client, which has to be explicitly done with the soap_send_fault operation when an error occurs.

In this way, multiple services can be chained to accept messages on the same port. This approach also works with SSL for HTTPS services.

However, this approach is not recommended for certain plugins, because plugins must be registered with all service objects and some plugins require state information to be used across the service objects, which will add significantly to the complexity.

When plugin complications arise, it is best to have all services share the same context. This means that soapcpp2 option -j should be used instead of option -i. Each service class has a pointer member

to a soap struct context. This member pointer should point to the same soap context.

With option -j and -q the code to chain the services is as follows, based on a single **struct** soap engine context:

```
struct soap *soap = soap_new();
Abc::soapABCService abc(soap); // generated with soapcpp2 -j -S -qAbc
Uvw::soapUVWService uvw(soap); // generated with soapcpp2 -j -S -qUvw
Xyz::soapXYZService xyz(soap); // generated with soapcpp2 -j -S -qXyz

soap_bind(soap, NULL, 8080, 100);
soap_accept(soap);
if (soap_begin_serve(soap))
   ... error
else if (abc.dispatch() == SOAP_NO_METHOD)
{
   if (uvw.dispatch() == SOAP_NO_METHOD)
   {
      if (xyz.dispatch() == SOAP_NO_METHOD)
         ... error
   }
}
soap_destroy(soap);
soap_end(soap);
soap_free(soap); // only safe when abc, uvw, xyz are also deleted
```

7.2.9 How to Generate WSDL Service Descriptions

The gSOAP stub and skeleton compiler soapcpp2 generates WSDL (Web Service Description Language) service descriptions and XML Schema files when processing a header file. The tool produces one WSDL file for a set of service operations, which must be provided. The names of the function prototypes of the service operations must use the same namespace prefix and the namespace prefix is used to name the WSDL file. If multiple namespace prefixes are used to define service operations, multiple WSDL files will be created and each file describes the set of service operations belonging to a namespace prefix.

In addition to the generation of the `ns.wsdl` file, a file with a namespace mapping table is generated by the gSOAP compiler. An example mapping table is shown below:

```
struct Namespace namespaces[] =
{
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
   {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
   {"xsi", "http://www.w3.org/2001/XMLSchema-instance", "http://www.w3.org/*/XMLSchema-
instance"},
   {"xsd", "http://www.w3.org/2001/XMLSchema", "http://www.w3.org/*/XMLSchema"},
   {"ns", "http://tempuri.org"},
   {NULL, NULL}
};
```

This file can be incorporated in the client/service application, see Section 10.4 for details on namespace mapping tables.

To deploy a Web service, copy the compiled CGI service application to the designated CGI directory of your Web server. Make sure the proper file permissions are set (`chmod 755 calc.cgi` for Unix/Linux). You can then publish the WSDL file on the Web by placing it in the appropriate Web server directory.

The gSOAP `soapcpp2` compiler also generates XML Schema files for all C/C++ complex types (e.g. **struct**s and **class**es) when declared with a namespace prefix. These files are named `ns.xsd`, where `ns` is the namespace prefix used in the declaration of the complex type. The XML Schema files do not have to be published as the WSDL file already contains the appropriate XML Schema definitions.

To customize the WSDL output, it is essential to use //`gsoap` directives to declare the service name, the endpoint port, and namespace:

```
//gsoap ns service name: example
//gsoap ns servire port: http://www.mydomain.com/example
//gsoap ns service namespace: urn:example
```

These are minimal settings. More details and settings for the service operations should be declared as well. See Section 19.2 for more details.

7.2.10   Example

For example, suppose the following methods are defined in the header file:

```
typedef double xsd__double;
int ns__add(xsd__double a, xsd__double b, xsd__double &result);
int ns__sub(xsd__double a, xsd__double b, xsd__double &result);
int ns__sqrt(xsd__double a, xsd__double &result);
```

Then, one WSDL file will be created with the file name `ns.wsdl` that describes all three service operations:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="Service"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://location/Service.wsdl"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:WSDL="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
  xmlns:tns="http://location/Service.wsdl"
  xmlns:ns="http://tempuri.org">
<types>
  <schema
```

```
        xmlns="http://www.w3.org/2000/10/XMLSchema"
        targetNamespace="http://tempuri.org"
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
        <complexType name="addResponse">
          <all>
            <element name="result" type="double" minOccurs="0" maxOccurs="1"/>
          </all>
          <anyAttribute namespace="##other"/>
        </complexType>
        <complexType name="subResponse">
          <all>
            <element name="result" type="double" minOccurs="0" maxOccurs="1"/>
          </all>
          <anyAttribute namespace="##other"/>
        </complexType>
        <complexType name="sqrtResponse">
          <all>
            <element name="result" type="double" minOccurs="0" maxOccurs="1"/>
          </all>
          <anyAttribute namespace="##other"/>
        </complexType>
      </schema>
</types>
<message name="addRequest">
  <part name="a" type="xsd:double"/>
  <part name="b" type="xsd:double"/>
</message>
<message name="addResponse">
  <part name="result" type="xsd:double"/>
</message>
<message name="subRequest">
  <part name="a" type="xsd:double"/>
  <part name="b" type="xsd:double"/>
</message>
<message name="subResponse">
  <part name="result" type="xsd:double"/>
</message>
<message name="sqrtRequest">
  <part name="a" type="xsd:double"/>
</message>
<message name="sqrtResponse">
  <part name="result" type="xsd:double"/>
</message>
<portType name="ServicePortType">
  <operation name="add">
    <input message="tns:addRequest"/>
    <output message="tns:addResponse"/>
  </operation>
  <operation name="sub">
    <input message="tns:subRequest"/>
```

```
      <output message="tns:subResponse"/>
    </operation>
    <operation name="sqrt">
      <input message="tns:sqrtRequest"/>
      <output message="tns:sqrtResponse"/>
    </operation>
</portType>
<binding name="ServiceBinding" type="tns:ServicePortType">
    <SOAP:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="add">
      <SOAP:operation soapAction="http://tempuri.org#add"/>
      <input>
        <SOAP:body use="encoded" namespace="http://tempuri.org"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <SOAP:body use="encoded" namespace="http://tempuri.org"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
    <operation name="sub">
      <SOAP:operation soapAction="http://tempuri.org#sub"/>
      <input>
        <SOAP:body use="encoded" namespace="http://tempuri.org"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <SOAP:body use="encoded" namespace="http://tempuri.org"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
    <operation name="sqrt">
      <SOAP:operation soapAction="http://tempuri.org#sqrt"/>
      <input>
        <SOAP:body use="encoded" namespace="http://tempuri.org"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <SOAP:body use="encoded" namespace="http://tempuri.org"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
</binding>
<service name="Service">
  <port name="ServicePort" binding="tns:ServiceBinding">
    <SOAP:address location="http://location/Service.cgi"/>
  </port>
</service>
</definitions>
```

The above uses all default settings for the service name, port, and namespace which should be set in the header file with //gsoap directives (Section 19.2).

7.2.11   How to Use Client Functionalities Within a Service

A gSOAP service implemented with CGI may make direct client calls to other services from within its service operations, without setting up a new context. A stand-alone service application must setup a new soap struct context, e.g. using soap_copy and delete it after the call.

The server-side client call is best illustrated with an example. The following example is a more sophisticated example that combines the functionality of two Web services into one new SOAP Web service. The service provides a currency-converted stock quote. To serve a request, the service in turn requests the stock quote and the currency-exchange rate from two XMethods services (these services are no longer available by XMethods, but are used here as an example).

In addition to being a client of two XMethods services, this service application can also be used as a client of itself to test the implementation. As a client invoked from the command-line, it will return a currency-converted stock quote by connecting to a copy of itself installed as a CGI application on the Web to retrieve the quote after which it will print the quote on the terminal.

The header file input to the gSOAP soapcpp2 compiler is given below. The example is for illustrative purposes only (the XMethods services are not operational):

```
// Contents of file "quotex.h":
int ns1__getQuote(char *symbol, float &result); // XMethods delayed stock quote service service
operation
int ns2__getRate(char *country1, char *country2, float &result); // XMethods currency-exchange
service service operation
int ns3__getQuote(char *symbol, char *country, float &result); // the new currency-converted
stock quote service
```

The quotex.cpp client/service application source is:

```
// Contents of file "quotex.cpp":
#include "soapH.h" // include generated proxy and SOAP support
int main(int argc, char **argv)
{
  struct soap soap;
  float q;
  soap_init(&soap);
  if (argc <= 2)
    soap_serve(&soap);
  else if (soap_call_ns3__getQuote(&soap, "http://www.cs.fsu.edu/\symbol{126}engelen/quotex.cgi",
"", argv[1], argv[2], q))
    soap_print_fault(&soap, stderr);
  else
    printf("\nCompany %s: %f (%s)\n", argv[1], q, argv[2]);
  return 0;
}
```

```
int ns3__getQuote(struct soap *soap, char *symbol, char *country, float &result)
{
  float q, r;
  int socket = soap->socket; // save socket (stand-alone service only, does not support keep-alive)
  if (soap_call_ns1__getQuote(soap, "http://services.xmethods.net/soap", "", symbol, &q) == 0 &&
      soap_call_ns2__getRate(soap, "http://services.xmethods.net/soap", NULL, "us", country, &r) == 0)
  {
    result = q*r;
    soap->socket = socket;
    return SOAP_OK;
  }
  soap->socket = socket;
  return SOAP_FAULT; // pass soap fault messages on to the client of this app
}
/* Since this app is a combined client-server, it is put together with
one header file that describes all service operations. However, as a consequence we
have to implement the methods that are not ours. Since these implementations are
never called (this code is client-side), we can make them dummies as below.
/
int ns1__getQuote(struct soap *soap, char *symbol, float &result)
{ return SOAP_NO_METHOD; } // dummy: will never be called
int ns2__getRate(struct soap *soap, char *country1, char *country2, float &result)
{ return SOAP_NO_METHOD; } // dummy: will never be called

struct Namespace namespaces[] =
{
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
  {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
  {"xsi", "http://www.w3.org/2001/XMLSchema-instance", "http://www.w3.org/*/XMLSchema-instance"},
  {"xsd", "http://www.w3.org/2001/XMLSchema", "http://www.w3.org/*/XMLSchema"},
  {"ns1", "urn:xmethods-delayed-quotes"},
  {"ns2", "urn:xmethods-CurrencyExchange"},
  {"ns3", "urn:quotex"},
  {NULL, NULL}
};
```

To compile:

```
> soapcpp2 quotex.h
> c++ -o quotex.cgi quotex.cpp soapC.cpp soapClient.cpp soapServer.cpp stdsoap2.cpp -lsocket
-lxnet -lnsl
```

Note: under Linux and Mac OS X you can often omit the -l libraries.

The quotex.cgi executable is installed as a CGI application on the Web by copying it in the designated directory specific to your Web server. After this, the executable can also serve to test the service. For example

```
> quotex.cgi IBM uk
```

returns the quote of IBM in uk pounds by communicating the request and response quote from the CGI application. See `http://xmethods.com/detail.html?id=5` for details on the currency abbreviations.

When combining clients and service functionalities, it is required to use one header file input to the compiler. As a consequence, however, stubs and skeletons are available for **all** service operations, while the client part will only use the stubs and the service part will use the skeletons. Thus, dummy implementations of the unused service operations need to be given which are never called.

Three WSDL files are created by gSOAP: `ns1.wsdl`, `ns2.wsdl`, and `ns3.wsdl`. Only the `ns3.wsdl` file is required to be published as it contains the description of the combined service, while the others are generated as a side-effect (and in case you want to develop these separate services).

7.3   Asynchronous One-Way Message Passing

SOAP RPC client-server interaction is synchronous: the client blocks until the server responds to the request. gSOAP also supports asynchronous one-way message passing and the interoperable synchronous one-way message passing over HTTP. The two styles are similar, but only the latter is interoperable and is compliant to Basic Profile 1.0. The interoperable synchronous one-way message passing style over HTTP is discussed in Section 7.4 below.

SOAP messaging routines are declared as function prototypes, just like service operations for SOAP RPC. However, the output parameter is a **void** type to indicate the absence of a return value.

For example, the following header file specifies an event message for SOAP messaging:

```
int ns__event(int eventNo, void);
```

The gSOAP `soapcpp2` tool generates the following functions in `soapClient.cpp`:

```
int soap_send_ns__event(struct soap *soap, const char URL, const char action, int event);
int soap_recv_ns__event(struct soap *soap, struct ns__event *dummy);
```

The `soap_send_ns__event` function transmits the message to the destination URL by opening a socket and sending the SOAP encoded message. The socket will remain open after the send and has to be closed with `soap_closesock()`. The open socket connection can also be used to obtain a service response, e.g. with a `soap_recv` function call.

The `soap_recv_ns__event` function waits for a SOAP message on the currently open socket (`soap.socket`) and fills the **struct** `ns__event` with the `ns__event` parameters (e.g. **int** eventNo). The **struct** `ns__event` is automatically created by gSOAP and is a mirror image of the `ns__event` parameters:

```
struct ns__event
{ int eventNo;
}
```

The gSOAP generated `soapServer.cpp` code includes a skeleton routine to accept the message. (The skeleton routine does not respond with a SOAP response message.)

```
int soap_serve_ns__event(struct soap *soap);
```

The skeleton routine calls the user-implemented ns__event(**struct** soap *soap, **int** eventNo) routine (note the absence of the void parameter!).

As usual, the skeleton will be automatically called by the service operation request dispatcher that handles both the service operation requests (RPCs) and messages:

```
int main()
{ soap_serve(soap_new());
}
int ns__event(struct soap *soap, int eventNo)
{
    ... // handle event
    return SOAP_OK;
}
```

7.4 Implementing Synchronous One-Way Message Passing over HTTP

One-way SOAP message passing over HTTP as defined by the SOAP specification and Basic Profile 1.0 is synchrounous, meaning that the server must respond with an HTTP OK header (or HTTP 202 Accepted) and an empty body. To implement synchrounous one-way messaging, the same setup for asynchrounous one-way messaing discussed in Section 7.3 is used, but with one simple addition at the client and server side for HTTP transfer.

At the server side, we have to return an empty HTTP OK response. Normally with one-way messaging the gSOAP engine closes the socket when the service operation is finished, which is not desirable for synchronous one-way message exchanges over HTTP: an HTTP response should be send. This is accomplished as follows. For each one-way operation implemented in C/C++, we replace the **return** SOAP_OK with:

```
int ns__event(struct soap *soap, int eventNo)
{
    ... // handle event
    return soap_send_empty_response(soap, SOAP_OK); // SOAP_OK: return HTTP 202 ACCEPTED
}
```

At the client side, the empty response header must be parsed as follows:

```
if (soap_send_ns__event(soap, eventNo) != SOAP_OK
    || soap_recv_empty_response(soap) != SOAP_OK)
    soap_print_fault(soap, stderr);
...
```

The synchronous (and asynchronous) one-way messaging supports HTTP keep-alive and chunking.

Note: soap_send_empty_response returns the error code SOAP_STOP to force the engine to stop producing a response message after the service operation completed, which allows soap_send_empty_response to be used with any service operation that should return HTTP 202.

66

7.5 How to Use the SOAP Serializers and Deserializers to Save and Load Application Data using XML Data Bindings

The gSOAP XML databindings for C and C++ allow a seamless integration of XML in C and C++ applications. Data can be serialized in XML and vice versa. WSDL and XML schema files can be converted to C or C++ definitions. C and C++ definitions can be translated to WSDL and schemas to support legacy ANSI C applications for example.

7.5.1 Mapping XML Schema to C/C++ with wsdl2h

Command:

> wsdl2h [options] XSD and WSDL files ...

The WSDL 1.1 and 2.0 standards are supported. If you have trouble with WSDL 2.0 please contact the author. The entire XML schema 1.1 standard is supported, except XPath expressions and assertions. This covers all of the following schema components with their optional [attributes] shown:

```
<xs:any [minOccurs, maxOccurs] >
<xs:anyAttribute>
<xs:all>
<xs:choice [minOccurs, maxOccurs] >
<xs:sequence [minOccurs, maxOccurs] >
<xs:group [name, ref] >
<xs:attributeGroup [name, ref] >
<xs:attribute [name, ref, type, use, default, fixed, form, wsdl:arrayType] >
<xs:element [name, ref, type, default, fixed, form, nillable, abstract,
substitutionGroup, minOccurs, maxOccurs] >
<xs:simpleType [name] >
<xs:complexType [name, abstract, mixed] >
```

The supported facets are:

```
<xs:enumeration>
<xs:simpleContent>
<xs:complexContent>
<xs:list>
<xs:extension>
<xs:restriction>
<xs:length>
<xs:minLength>
<xs:maxLength>
<xs:minInclusive>
<xs:maxInclusive>
<xs:minExclusive>
<xs:maxExclusive>
<xs:precision>    maps to float/double with C formatted output
<xs:scale>    maps to float/double with C formatted output
```

```
<xs:totalDigits>     maps to float/double with C formatted output
<xs:fractionDigits>      maps to float/double with C formatted output
<xs:pattern>     content not automatically validated yet
<xs:union>     maps to string, content not validated yet
```

Other:

```
<xs:import>
<xs:include>
<xs:redefine>
<xs:annotation>
```

All primitive XSD types are supported (with the default mapping shown):

```
xsd:string      maps to string (char*,wchar_t*,std::string,std::wstring)
xsd:boolean      maps to bool (C++) or enum xsd__boolean (C)
xsd:float      maps to float
xsd:double      maps to double
xsd:decimal      maps to string, or use "#import "custom/decimal.h"
xsd:precisionDecimal      maps to string
xsd:duration      maps to string, or use "#import "custom/duration.h"
xsd:dateTime      maps to time_t, or use "#import "custom/struct_tm.h"
xsd:time      maps to string
xsd:date      maps to string
xsd:gYearMonth      maps to string
xsd:gYear      maps to string
xsd:gMonth      maps to string
xsd:hexBinary      maps to struct xsd__hexBinary
xsd:base64Bianry      maps to struct xsd__base64Binary
xsd:anyURI      maps to string
xsd:QName      maps to _QName (string normalization rules apply)
xsd:NOTATION      maps to string
```

Note: string targets are defined in the typemap.dat file used by wsdl2h to map XSD types. This allows the use of **char***, **wsha_t***, std::string, and std::wstring string types for all XSD types mapped to strings.

All non-primitive XSD types are supported (with the default mapping shown):

```
xsd:normalizedString      maps to string
xsd:token      maps to string
xsd:language      maps to string
xsd:IDREFS      maps to string
xsd:ENTITIES      maps to string
xsd:NMTOKEN      maps to string
xsd:NMTOKENS      maps to string
xsd:Name      maps to string
xsd:NCName      maps to string
xsd:ID      maps to string
xsd:IDREF      maps to string
xsd:ENTITY      maps to string
```

```
xsd:integer    maps to string
xsd:nonPositiveInteger    maps to string
xsd:negativeInteger    maps to string
xsd:long    maps to LONG64
xsd:int    maps to int
xsd:short    maps to short
xsd:byte    maps to byte
xsd:nonNegativeInteger    maps to string
xsd:unsignedLong    maps to ULONG64
xsd:unsignedInt    maps to unsigned int
xsd:unsignedShort    maps to unsigned short
xsd:unsignedByte    maps to unsigned byte
xsd:positiveInteger    maps to string
xsd:yearMonthDuration    maps to string
xsd:dayTimeDuration    maps to string
xsd:dateTimeStamp    maps to string
```

There are several initialization flags to control XML serialization at runtime:

- XML content validation is enforced with SOAP_XML_STRICT.

- XML namespaces are supported, unless disabled with SOAP_XML_IGNORENS.

- XML exclusive canonicalization is enabled with SOAP_XML_CANONICAL.

- XML default xmlns="..." namespace bindings are used with SOAP_XML_DEFAULTNS.

- XML is indented for enhanced readability with SOAP_XML_INDENT.

- XML xsi:nil for NULL elements is serialized with SOAP_XML_NIL.

To obtain C and/or C++ type definitions for XML schema components, run wsdl2h on the schemas to generate a header file. This header file defines the C/C++ type representations of the XML schema components. The header file is then processed by the soapcpp2 tool to generate the serializers for these types. See Section 1.4 for an overview to use wsdl2h and soapcpp2 to map schemas to C/C++ types to obtain XML data bindings.

7.5.2   Mapping C/C++ to XML Schema with soapcpp2

To generate serialization code, execute:

> soapcpp2 [options] header_file.h

The following C/C++ types are supported in the header file:

bool
enum, **enum*** ('**enum***' indicates serialized as a bitmask)
(**unsigned**) **char**, **short**, **int**, **long**, **long long** (also LONG64), **size_t**
float, **double**, **longdouble**(#import "custom/long_double.h")
std::string, std::wstring, **char[]**, **char***, **wchar_t***

69

_XML (a char* type to hold literal XML string content)
_QName (a char* type with normalized QName content of the form prefix:name)
struct, **class** (with single inheritance)
std::vector, std::list, std::deque, std::set (#import "import/stl.h")
union (requires preceding discriminant member field)
typedef
time_t
template<> **class**(requires begin(), end(), size(), and insert() methods)
void* (requires a preceding _ _type field to indicate the object pointed to)
struct xsd_ _hexBinary (special pre-defined type to hold binary content)
struct xsd_ _base64Binary (special pre-defined type to hold binary content)
struct tm (#import "custom/struct_tm.h")
struct timeval (#import "custom/struct_timeval.h")
pointers to any of the above (any pointer-linked structures are serializable, including cyclic graphs)
fixed-size arrays of all of the above

Additional features and potential limitations:

- A header file should not include any code statements, only data type declarations.

- Nested classes and nested types are unnested.

- Use #import "file.h" instead of #include to import other header files. The #include and #define directives are accepted, but deferred to the generated code.

- C++ namespaces are supported (must cover entire header file content)

- Optional DOM support can be used to store mixed content or literal XML content. Otherwise, mixed content may be lost. Use soapcpp2 option -d for DOM support.

- Types are denoted transient using the '**extern**' qualifier, which prevents serialization as desired:

    extern class name; // class 'name' is not serialized
    struct name { **extern char** *name; **int** num; }; // 'name' is not serialized

- Only public members of a class can be serialized:

    class name { **private**: **char** *secret; }; // 'secret' is not serialized

- Types are denoted "volatile", which means that they are declared elsewhere in the source code and should not be redeclared in the generated code nor augmented by the soapcpp2 tool:

    volatile class name { ... }; // defined here just to generate the serializers

- struct/class members are serialized as attributes when qualified with '@':

    struct record { **@char** *name; **int** num; }; // attribute name, element num

- Strings with 8-bit content can hold ASCII (default) or UTF8. The latter is possible by enabling the SOAP_C_UTFSTRING flag. When enabled, all std::string and **char*** strings MUST contain UTF8.

70

The soapcpp2 tool generates serializers and deserializers for all wsdl2h-generated or user-defined data structures that are specified in the header file input to the compiler. The serializers and deserializers can be found in the generated soapC.cpp file. These serializers and deserializers can be used separately by an application without the need to build a full client or service application. This is useful for applications that need to save or export their data in XML or need to import or load data stored in XML format.

7.5.3   Serializing C/C++ Data to XML

We assume that the wsdl2h tool was used to map XML schema types to C/C++ data types. The soapcpp2 tool then generates the (de)serializers for the C/C++ types. You can also use soapcpp2 directly on a header file that declares annotated C/C++ data types to serialize.

The following attributes can be set to control the destination and source for serialization and deserialization:

Variable	Description
int soap.socket	socket file descriptor for input and output (or set to SOAP_INVALID_SOCKET)
ostream *soap.os	(C++ only) output stream used for send operations
istream *soap.is	(C++ only) input stream used for receive operations
int soap.sendfd	when soap.socket=SOAP_INVALID_SOCKET, this fd is used for send operations
int soap.recvfd	when soap.socket=SOAP_INVALID_SOCKET, this fd is used for receive operations

The following initializing and finalizing functions can be used:

Function	Description
void soap_begin_send(**struct** soap*)	start a send/write phase
int soap_end_send(**struct** soap*)	flush the buffer
int soap_begin_recv(**struct** soap*)	start a rec/read phase (if an HTTP header is present, parse it first)
int soap_end_recv(**struct** soap*)	perform a id/href consistency check on deserialized data

These operations do not open or close the connections. The application should open and close connections or files and set the soap.socket, soap.os or soap.sendfd, soap.is or soap.recvfd streams or descriptors. When soap.socket<0 and none of the streams and descriptors are set, then the standard input and output will be used.

The following options are available to control serialization:

    soap-¿encodingStyle = NULL; // to remove SOAP 1.1/1.2 encodingStyle
    soap_mode(soap, SOAP_XML_TREE); // XML without id-ref (no cycles!)
    soap_mode(soap, SOAP_XML_GRAPH); // XML with id-ref (including cycles)
    soap_set_namespaces(soap, **struct** Namespace *nsmap); //to set xmlns bindings

See also Section 9.12 to control the I/O buffering and content encoding such as compression and DIME encoding.

We assume that the wsdl2h tool was used to map XML schema types to C/C++ data types. The soapcpp2 tool then generates the (de)serializers for the C/C++ types.

To serialize data to an XML stream, two functions should be called to prepare for serialization of the data and to send the data, respectively. The first function, soap_serialize, analyzes pointers and

determines if multi-references are required to encode the data and if cycles are present the object graph. The second function, soap_put, produces the XML output on a stream.

The soap_serialize and soap_put (and both combined by soap_write) functions are statically generated specific to a data type. For example, soap_serialize_float(&soap, &d) is called to serialize an **float** value and soap_put_float(&soap, &d, "number", NULL) is called to output the floating point value in SOAP tagged with the name <number>. The soap_write_float(&soap, &d) conveniently combines the initialization of output, writing the data, and finalizing the output.

To initialize data, the soap_default function of a data type can be used. For example, soap_default_float(&soap, &d) initializes the float to 0.0. The soap_default functions are useful to initialize complex data types such as arrays, **struct**s, and **class** instances. Note that the soap_default functions do not need the gSOAP runtime context as a first parameter.

The following table lists the type naming conventions used by gSOAP:

Type	Type Name
char*	string
wchar_t*	wstring
char	byte
bool	bool
double	double
int	int
float	float
long	long
LONG64	LONG64 (Win32)
long long	LONG64 (Unix/Linux)
short	short
time_t	time
unsigned char	unsignedByte
unsigned int	unsignedInt
unsigned long	unsignedLong
ULONG64	unsignedLONG64 (Win32)
unsigned long long	unsignedLONG64 (Unix/Linux)
unsigned short	unsignedShort
$T[N]$	Array$N$Of$Type$ where $Type$ is the type name of T
$T*$	PointerTo$Type$ where $Type$ is the type name of T
struct Name	Name
class Name	Name
enum Name	Name

Consider for example the following C code with a declaration of p as a pointer to a **struct** ns__Person:

        struct ns__Person { **char** *name; } *p;

To serialize p, its address is passed to the function soap_serialize_PointerTons__Person generated for this type by the gSOAP soapcpp2 compiler:

        soap_serialize_PointerTons__Person(&soap, &p);

The **address of** p is passed, so the serializer can determine whether p was already serialized and to discover co-referenced objects and cycles in graph data structures that require SOAP encod-

ing with id-ref serialization. To generate the output, the address of p is passed to the function soap_put_PointerTons_ _Person together with the name of an XML element and an optional type string (to omit a type, use NULL):

```
soap_begin_send(&soap);
soap_put_PointerTons_ _Person(&soap, &p, "ns:element-name", "ns:type-name");
soap_end_send(&soap);
```

or the shorthand for the above (without the xsi type):

```
soap_write_PointerTons_ _Person(&soap, &p);
```

This produces:

```
<ns:element-name xmlns:SOAP-ENV="..." xmlns:SOAP-ENC="..." xmlns:ns="..."
   ...  xsi:type="ns:type-name">
<name xsi:type="xsd:string">...</name>
</ns:element-name>
```

The serializer is initialized with the soap_begin_send(soap) function and closed with soap_end_send(soap). All temporary data structures and data structures deserialized on the heap are destroyed with the soap_destroy and soap_end functions (in this order).

The soap_done function should be used to reset the context, i.e. the last use of the context. To detach and deallocate the context, use soap_free.

To remove the temporary data only and keep the deserialized data on the heap, use soap_free_temp. Temporary data structures are only created if the encoded data uses pointers. Each pointer in the encoded data has an internal hash table entry to determine all multi-reference parts and cyclic parts of the complete data structure.

You can assign an output stream to soap.os or a file descriptor to soap.sendfd. For example

```
soap.sendfd = open(file, O_RDWR|O_CREAT, S_IWUSR|S_IRUSR);
soap_serialize_PointerTons_ _Person(&soap, &p);
soap_begin_send(&soap);
soap_put_PointerTons_ _Person(&soap, &p, "ns:element-name", "ns:type-name");
soap_end_send(&soap);
```

The above can be abbreviated to

```
soap.sendfd = open(file, O_RDWR|O_CREAT, S_IWUSR|S_IRUSR);
soap_write_PointerTons_ _Person(&soap, &p);
```

The soap_serialize function is optional. It MUST be used when the object graph contains cycles. It MUST be called to preserve the logical coherence of pointer-based data structures, where pointers may refer to co-referenced objects. By calling soap_serialize, data structures shared through pointers are serialized only once and referenced in XML using id-refs attributes. The actual id-refs used depend on the SOAP encoding. To turn off SOAP encoding, remove or avoid using the SOAP-ENV and SOAP-ENC namespace bindings in the namespace table. In addition, the SOAP_XML_TREE and

SOAP_XML_GRAPH flags can be used to control the output by restricting serialization to XML trees or by enabling multi-ref graph serialization with id-ref attribuation.

To save the data as an XML tree (with one root) without any id-ref attributes, use the SOAP_XML_TREE flag. The data structure MUST NOT contain pointer-based cycles.

To preserve the exact structure of the data object graph and create XML with one root, use the SOAP_XML_GRAPH output-mode flag (see Section 9.12). Use this flag and the soap_serialize function to prepare the serialization of data with in-line id-ref attributes. Using the SOAP_XML_GRAPH flag assures the preservation of the logical structure of the data

For example, to encode the contents of two variables var1 and var2 that may share data through pointer structures, the serializers are called before the output routines:

```
T1 var1;
T2 var2;
struct soap soap;
...
soap_init(&soap); // initialize
[soap_omode(&soap, flags);] // set output-mode flags (e.g. SOAP_ENC_XML|SOAP_ENC_ZLIB)
soap_begin(&soap); // start new (de)serialization phase
soap_set_omode(&soap, SOAP_XML_GRAPH);
soap_serialize_Type1(&soap, &var1);
soap_serialize_Type2(&soap, &var2);
...
[soap.socket = a_socket_file_descriptor;] // when using sockets
[soap.os = an_output_stream;] // C++
[soap.sendfd = an_output_file_descriptor;] // C
soap_begin_send(&soap);
soap_put_Type1(&soap, &var1, "[namespace-prefix:]element-name1", "[namespace-prefix:]type-name1");
soap_put_Type2(&soap, &var2, "[namespace-prefix:]element-name2", "[namespace-prefix:]type-name2");
...
soap_end_send(&soap); // flush
soap_destroy(&soap); // remove deserialized C++ objects
soap_end(&soap); // remove deserialized data structures
soap_done(&soap); // finalize last use of this context
...
```

where Type1 is the type name of T1 and Type2 is the type name of T2 (see table above). The strings [namespace-prefix:]type-name1 and [namespace-prefix:]type-name2 describe the schema types of the elements. Use NULL to omit this type information.

For serializing class instances, method invocations MUST be used instead of function calls, for example obj.soap_serialize(&soap) and obj.soap_put(&soap, "elt", "type"). This ensures that the proper serializers are used for serializing instances of derived classes.

You can serialize a class instance to a stream as follows:

```
struct soap soap;
myClass obj;
soap_init(&soap); // initialize
soap_begin(&soap); // start new (de)serialization phase
```

```
soap_set_omode(&soap, SOAP_XML_GRAPH);
obj.serialize(&soap);
soap.os = &cout; // send to cout
soap_begin_send(&soap);
obj.put(&soap, "[namespace-prefix:]element-name1", "[namespace-prefix:]type-name1");
soap_end_send(&soap); // flush
...
soap_destroy(&soap); // remove deserialized C++ objects
soap_end(&soap); // remove deserialized data
soap_done(&soap); // finalize last use of this context
```

When you declare a soap struct pointer as a data member in a class, you can overload the `<<` operator to serialize the class to streams:

```
ostream &operator<<(ostream &o, const myClass &e)
{
  if (!e.soap)
  ... error: need a soap struct to serialize (could use global struct) ...
  else
  {
    ostream *os = e.soap->os;
    e.soap->os = &o;
    soap_set_omode(e.soap, SOAP_XML_GRAPH);        e.serialize(e.soap);
    soap_begin_send(e.soap);
    e.put(e.soap, "myClass", NULL);
    soap_end_send(e.soap);
    e.soap->os = os;
    soap_clr_omode(e.soap, SOAP_XML_GRAPH);
  }
  return o;
}
```

Of course, when you construct an instance you must set its soap struct to a valid context. Deserialized class instances with a soap struct data member will have their soap structs set automatically, see Section 9.13.2.

In principle, XML output for a data structure can be produced with soap_put without calling the soap_serialize function first. In this case, the result is similar to SOAP_XML_TREE which means that no id-refs are output. Cycles in the data structure will crash the serialization algorithm, even when the SOAP_XML_GRAPH is set.

Consider the following **struct**:

```
// Contents of file "tricky.h":
struct Tricky
{
  int *p;
  int n;
  int *q;
};
```

The following fragment initializes the pointer fields p and q to the value of field n:

```
struct soap soap;
struct Tricky X;
X.n = 1;
X.p = &X.n;
X.q = &X.n;
soap_init(&soap);
soap_begin(&soap);
soap_serialize_Tricky(&soap, &X);
soap_put_Tricky(&soap, &X, "Tricky", NULL);
soap_end(&soap); // Clean up temporary data used by the serializer
```

What is special about this data structure is that n is 'fixed' in the Tricky structure, and p and q both point to n. The gSOAP serializers strategically place the id-ref attributes such that n will be identified as the primary data source, while p and q are serialized with ref/href attributes.

The resulting output is:

```
<Tricky xsi:type="Tricky">
<p href="#2"/> <n xsi:type="int">1</n> <q href="#2"/> <r xsi:type="int">2</r> </Tricky>
<id id="2" xsi:type="int">1</id>
```

which uses an independent element at the end to represent the multi-referenced integer, assuming the SOAP-ENV and SOAP-ENC namespaces indicate SOAP 1.1 encoding.

With the SOAP_XML_GRAPH flag the output is:

```
<Tricky xsi:type="Tricky">
<p href="#2"/> <n id="2" xsi:type="int">1</n> <q href="#2"/> </Tricky>
```

In this case, the XML is self-contained and multi-referenced data is accurately serialized. The gSOAP generated deserializer for this data type will be able to accurately reconstruct the data from the XML (on the heap).

7.5.4 Deserializing C/C++ Data from XML

We assume that the wsdl2h tool was used to map XML schema types to C/C++ data types. The soapcpp2 tool then generates the (de)serializers for the C/C++ types. You can also use soapcpp2 directly on a header file that declares annotated C/C++ data types to serialize.

To deserialize a data type from XML, the soap_get (or the simpler soap_read) function for the data type to be deserialized is used. The outline of a program that deserializes two variables var1 and var2 is for example:

```
T1 var1;
T2 var2;
struct soap soap;
...
soap_init(&soap); // initialize at least once
[soap_imode(&soap, flags);] // set input-mode flags
soap_begin(&soap); // begin new decoding phase
```

```
[soap.is = an_input_stream;] // C++
[soap.recvfd = an_input_file_desriptpr;] // C
soap_begin_recv(&soap); // if HTTP/MIME/DIME/GZIP headers are present, parse them
if (!soap_get_Type1(&soap, &var1, "[namespace-prefix:]element-name1", "[namespace-prefix:]type-
name1"))
    ... error ...
if (!soap_get_Type2(&soap, &var2, "[namespace-prefix:]element-name2", "[namespace-prefix:]type-
name1"))
    ... error ...
...
soap_end_recv(&soap); // check consistency of id/hrefs
soap_destroy(&soap); // remove deserialized C++ objects
soap_end(&soap); // remove deserialized data
soap_done(&soap); // finalize last use of the context
```

The strings [namespace-prefix:]type-name1 and [namespace-prefix:]type-name2 are the schema types of the elements and should match the `xsi:type` attribute of the receiving message. To omit the match, use NULL as the type. For class instances, method invocation can be used instead of a function call if the object is already instantiated, i.e. obj.soap_get(&soap, "...", "...").

The soap_begin call resets the deserializers. The soap_destroy and soap_end calls remove the temporary data structures **and** the decoded data that was placed on the heap.

To remove temporary data while retaining the deserialized data on the heap, the function soap_free_temp should be called instead of soap_destroy and soap_end.

One call to the soap_get_Type function of a type Type scans the entire input to process its XML content and to capture SOAP 1.1 independent elements (which contain multi-referenced objects). As a result, soap.error will set to SOAP_EOF. Also storing multiple objects into one file will fail to decode them properly with multiple soap_get calls. A well-formed XML document should only have one root anyway, so don't save multiple objects into one file. If you must save multiple objects, create a linked list or an array of objects and save the linked list or array. You could use the soap_in_Type function instead of the soap_get_Type function. The soap_in_Type function parses one XML element at a time.

You can deserialize class instances from a stream as follows:

```
myClass obj;
struct soap soap;
soap_init(&soap); // initialize
soap.is = &cin; // read from cin
soap_begin_recv(&soap); // if HTTP header is present, parse it
if (soap_get_myClass(&soap, &obj, "myClass", NULL) == NULL)
    ... error ...
soap_end_recv(&soap); // check consistency of id/hrefs
...
soap_destroy(&soap); // remove deserialized C++ objects
soap_end(&soap); // remove deserialized data
soap_done(&soap); // finalize last use of the context
```

This can be abbreviated to:

77

```
myClass obj;
struct soap soap;
soap_init(&soap); // initialize
soap.is = &cin; // read from cin
if (soap_read_myClass(&soap, &obj, NULL) != SOAP_OK)
    ... error ...
...
soap_destroy(&soap); // remove deserialized C++ objects
soap_end(&soap); // remove deserialized data
soap_done(&soap); // finalize last use of the context
```

When declaring a soap struct pointer as a data member in a class, you can overload the **>>** operator
to parse and deserialize a class instance from a stream:

```
istream &operator>>(istream &i, myClass &e)
{
    if (!e.soap)
    ... error: need soap struct to deserialize (could use global struct)...
    istream *is = e.soap->is;
    e.soap->is = &i;
    if (soap_read_myClass(e.soap, &e) != SOAP_OK)
        ... error ...
    e.soap->is = is;
    return i;
}
```

7.5.5   Example

As an example, consider the following data type declarations:

```
// Contents of file "person.h":
typedef char *xsd__string;
typedef char *xsd__Name;
typedef unsigned int xsd__unsignedInt;
enum ns__Gender {male, female};
class ns__Address
{
    public:
    xsd__string street;
    xsd__unsignedInt number;
    xsd__string city;
};
class ns__Person
{
    public:
    xsd__Name name;
    enum ns__Gender gender;
    ns__Address address;
    ns__Person *mother;
    ns__Person *father;
};
```

The following program uses these data types to write to standard output a data structure that contains the data of a person named "John" living at Downing st. 10 in Londen. He has a mother "Mary" and a father "Stuart". After initialization, the class instance for "John" is serialized and encoded in XML to the standard output stream using gzip compression (requires the Zlib library, compile sources with -DWITH_GZIP):

```cpp
// Contents of file "person.cpp":
#include "soapH.h"
int main()
{
    struct soap soap;
    ns__Person mother, father, john;
    mother.name = "Mary";
    mother.gender = female;
    mother.address.street = "Downing st.";
    mother.address.number = 10;
    mother.address.city = "London";
    mother.mother = NULL;
    mother.father = NULL;
    father.name = "Stuart";
    father.gender = male;
    father.address.street = "Main st.";
    father.address.number = 5;
    father.address.city = "London";
    father.mother = NULL;
    father.father = NULL;
    john.name = "John";
    john.gender = male;
    john.address = mother.address;
    john.mother = &mother;
    john.father = &father;
    soap_init(&soap);
    soap_omode(&soap, SOAP_ENC_ZLIB|SOAP_XML_GRAPH); // see 9.12
    soap_begin(&soap);
    soap_begin_send(&soap);
    john.soap_serialize(&soap);
    john.soap_put(&soap, "johnnie", NULL);
    soap_end_send(&soap);
    soap_destroy(&soap);
    soap_end(&soap);
    soap_done(&soap);
}
struct Namespace namespaces[] =
{
    {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
    {"SOAP-ENC","http://schemas.xmlsoap.org/soap/encoding/"},
    {"xsi", "http://www.w3.org/2001/XMLSchema-instance"},
    {"xsd", "http://www.w3.org/2001/XMLSchema"},
    {"ns", "urn:person"}, // Namespace URI of the "Person" data type
    {NULL, NULL}
};
```

The header file is processed and the application compiled on Linux/Unix with:

```
> soapcpp2 person.h
> c++ -DWITH_GZIP -o person person.cpp soapC.cpp stdsoap2.cpp -lsocket -lxnet -lnsl -lz
```

(Depending on your system configuration, the libraries libsocket.a, libxnet.a, libnsl.a are required. Compiling on Linux typically does not require the inclusion of those libraries.) See 19.27 for details on compression with gSOAP.

Running the person application results in the compressed XML output:

```
<johnnie xsi:type="ns:Person" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns="urn:person"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<name xsi:type="xsd:Name">John</name>
<gender xsi:type="ns:Gender">male</gender>
<address xsi:type="ns:Address">
<street id="3" xsi:type="xsd:string">Dowling st.</street>
<number xsi:type="unsignedInt">10</number>
<city id="4" xsi:type="xsd:string">London</city>
</address>
<mother xsi:type="ns:Person">
<name xsi:type="xsd:Name">Mary</name>
<gender xsi:type="ns:Gender">female</gender>
<address xsi:type="ns:Address">
<street href="#3"/>
<number xsi:type="unsignedInt">5</number>
<city href="#4"/>
</address>
</mother>
<father xsi:type="ns:Person">
<name xsi:type="xsd:Name">Stuart</name>
<gender xsi:type="ns:Gender">male</gender>
<address xsi:type="ns:Address">
<street xsi:type="xsd:string">Main st.</street>
<number xsi:type="unsignedInt">13</number>
<city href="#4"/>
</address>
</father>
</johnnie>
```

The following program fragment decodes this content from standard input and reconstructs the original data structure on the heap:

```
#include "soapH.h"
int main()
{
  struct soap soap;
```

```
      ns_ _Person *mother, *father, *john = NULL;
      soap_init(&soap);
      soap_imode(&soap, SOAP_ENC_ZLIB); // optional: gzip is detected automatically
      soap_begin(&soap);
      if ((john = soap_get_ns_ _Person(&soap, NULL, NULL, NULL)) == NULL)
         ... error ...
      mother = john->mother;
      father = john->father;
      ...
      soap_end_recv(&soap);
      soap_free_temp(&soap); // Clean up temporary data but keep deserialized data
   }
   struct Namespace namespaces[] =
   {
      {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
      {"SOAP-ENC","http://schemas.xmlsoap.org/soap/encoding/"},
      {"xsi", "http://www.w3.org/2001/XMLSchema-instance"},
      {"xsd", "http://www.w3.org/2001/XMLSchema"},
      {"ns", "urn:person"}, // Namespace URI of the "Person" data type
      {NULL, NULL}
   };
```

It is REQUIRED to either pass NULL to the soap_get routine, or a valid pointer to a data structure that can hold the decoded content. If the data john was already allocated then it does not need to be allocated again as the following demonstrates. The following program fragment decodes the SOAP content in a **struct** ns_ _Person allocated on the stack:

```
   #include "soapH.h"
   int main()
   {
      struct soap soap;
      ns_ _Person *mother, *father, john;
      soap_init(&soap);
      soap_default_ns_ _Person(&soap, &john);
      soap_imode(&soap, SOAP_ENC_ZLIB); // optional
      soap_begin(&soap);
      soap_begin_recv(&soap);
      if (soap_get_ns_ _Person(&soap, &john, "johnnie", NULL) == NULL)
         ... error ...
      ...
   }
   struct Namespace namespaces[] =
      ...
```

Note the use of soap_default_ns_ _Person. This routine is generated by the gSOAP soapcpp2 tool and assigns default values to the fields of john.

7.5.6 Serializing and Deserializing Class Instances to Streams

C++ applications can define appropriate stream operations on objects for (de)serialization of objects on streams. This is best illustrated with an example. Section 7.5.3 gives details on serializing types in general. Consider the class

```
class ns_ _person
{ public:
  char *name;
  struct soap *soap; // we need this, see below
  ns_ _person();
  ~ns_ _person();
};
```

The **struct** soap member is used to bind the instances to a gSOAP context for (de)serialization. We use the gSOAP soapcpp2 compiler from the command prompt to generate the class (de)serializers (assuming that person.h contains the class declaration):

```
> soapcpp2 person.h
```

gSOAP generates the (de)serializers and an instantiation function for the class soap_new_ns_ _person(**struct** soap *soap, **int** num) to instantiate one or more objects and associate them with a gSOAP context for deallocation with soap_destroy(soap). To instantiate a single object, omit the num parameter or set to -1. To instantiate an array of objects, set num$\geq$ 0.

```
#include "soapH.h"
#include "ns.nsmap"
...
struct soap *soap = soap_new();
ns_ _person *p = soap_new_ns_ _person(soap);
...
cout << p; // serialize p in XML
...
in >> p; // parse XML and deserialize p
...
soap_destroy(soap); // deletes p too
soap_end(soap);
soap_done(soap);
```

The stream operations are implemented as follows

```
ostream &operator<<(ostream &o, const ns_ _person &p)
{
  if (!p.soap)
    return o; // need a gSOAP context to serialize
  p.soap->os = &o;
  soap_omode(p.soap, SOAP_XML_TREE); // XML tree or graph
  p.soap_serialize(p.soap);
  soap_begin_send(p.soap);
  if (p.soap_put(p.soap, "person", NULL)
```

```
        || soap_end_send(p.soap))
      ; // handle I/O error
    return o;
}
istream &operator>>(istream &i, ns__person &p)
{
    if (!p.soap)
      return o; // need a gSOAP context to parse XML and deserialize
    p.soap->is = &i;
    if (soap_begin_recv(p.soap)
        || p.soap_in(p.soap, NULL, NULL)
        || soap_end_recv(p.soap))
      ; // handle I/O error
    return i;
}
```

7.5.7 How to Specify Default Values for Omitted Data

The gSOAP soapcpp2 compiler generates soap_default functions for all data types. The default values of the primitive types can be easily changed by defining any of the following macros in the stdsoap2.h file:

```
#define SOAP_DEFAULT_bool
#define SOAP_DEFAULT_byte
#define SOAP_DEFAULT_double
#define SOAP_DEFAULT_float
#define SOAP_DEFAULT_int
#define SOAP_DEFAULT_long
#define SOAP_DEFAULT_LONG64
#define SOAP_DEFAULT_short
#define SOAP_DEFAULT_string
#define SOAP_DEFAULT_time
#define SOAP_DEFAULT_unsignedByte
#define SOAP_DEFAULT_unsignedInt
#define SOAP_DEFAULT_unsignedLong
#define SOAP_DEFAULT_unsignedLONG64
#define SOAP_DEFAULT_unsignedShort
#define SOAP_DEFAULT_wstring
```

Instead of adding these to stdsoap2.h, you can also compile with option -DWITH_SOAPDEFS_H and include your definitions in file userdefs.h. The absence of a data value in a receiving SOAP message will result in the assignment of a default value to a primitive type upon deserialization.

Default values can also be assigned to individual **struct** and **class** fields of primitive type. For example,

```
struct MyRecord
{
    char *name = "Unknown";
    int value = 9999;
```

```
        enum Status { active, passive } status = passive;
    }
```

Default values are assigned to the fields on receiving a SOAP/XML message in which the data values are absent.

Because method requests and responses are essentially structs, default values can also be assigned to method parameters. The default parameter values do not control the parameterization of C/C++ function calls, i.e. all actual parameters must be present when calling a function. The default parameter values are used in case an inbound request or response message lacks the XML elements with parameter values. For example, a Web service can use default values to fill-in absent parameters in a SOAP/XML request:

int ns_ _login(**char** *uid = "anonymous", **char** *pwd = "guest", **bool** granted);

When the request message lacks uid and pwd parameters, e.g.:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns="http://tempuri.org">
  <SOAP-ENV:Body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <ns:login>
    </ns:login>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

then the service uses the default values. In addition, the default values will show up in the SOAP/XML request and response message examples generated by the gSOAP compiler.

8   The wsdl2h WSDL and Schema Importer

The wsdl2h tool is an advanced application that converts one or more WSDLs to C/C++. It can also be used without WSDLs to convert XML schemas (XSD files) to C/C++ to implement XML data bindings in C and C++.

The creation of C and C++ applications from one of more WSDL service descriptions is a two-step process.

To convert a WSDL to C++, use:

> wsdl2h file.wsdl

to generate a C++ header file file.h. This generated header file is a Web service specification that contains the parameter types and service function definitions in an understandable format in C++ (or ANSI C as shown below). Web service operations are represented as function prototypes.

Schema types are represented by semantically equivalent C/C++ types that are convenient and natural to use in a C/C++ application. The generated header file also contains various annotations related to the Web service properties defined in the WSDL.

To generate ANSI C, use option -c:

> wsdl2h -c <u>file</u>.wsdl

Multiple WSDL specifications can be processed at once and saved to one file with the -o option:

> wsdl2h -o file.h file1.wsdl file2.wsdl file3.wsdl

You can retrieve WSDLs from one of more URLs:

> wsdl2h -o file.h http://www.example.com/example.wsdl

To convert XML schemas to C or C++ XML data binding code, use:

> wsdl2h -o file.h file1.xsd file2.xsd file3.xsd

The wsdl2h-generated header file file.h is processed by the soapcpp2 tool to auto-generate the advanced data binding logic to convert the C/C++ data to XML and vice versa at runtime for your SOAP/XML application.

To process a gSOAP header file file.h (generated by wsdl2h) to generate advanced XML data bindings for C++, use:

> soapcpp2 -i -Iimport file.h

When the header file file.h was generated for C++, then this command generates a couple of C++ source files (more details will follow in Section 9) that implement XML encoders for the data binding. Option -i generates a client proxy objects and service objects to invoke and serve SOAP/XML operations, respectively. Option -Iimport sets the import directory for imported files from the package's import, such as stlvector.h for STL vector serialization support.

When the header file file.h was generated for ANSI C, then the above command generates a couple of C files that implement XML encoders, client stubs for remote invocation, and service skeletons for service operations.

Consider for example the following commands to implement a c++ client of a service:

> wsdl2h -o calc.h http://www.genivia.com/calc.wsdl
...
> soapcpp2 -i -Iimport calc.h

The first command generates calc.h from the WSDL at the specified URL. The header file is then processed by the soapcpp2 tool to generate the proxies (and service objects that we will not use) for the client application.

The C++ client application uses the auto-generated soapcalcProxy.h class and calc.nsmap XML namespace table to access the Web service. Both need to be #include-d in your source. Then compile and link the soapcalcProxy.cpp, soapC.cpp and stdsoap2.cpp sources to complete the build.

8.1 wsdl2h Options

The wsdl2h tool is an advanced XML data binding tool for converting WSDLs and XML schemas (XSD files) to C or C++. The tool takes WSDL and/or XSD files or URLs and converts these to a C or C++ specification in one easy-to-read C/C++ header file. **The header file is not intended to be included in your code directly!**. It should be converted by soapcpp2 to generate the logic for the data bindings. It can however be safely converted by a documentation tool such as Doxygen to analyze and represent the service operations and data in a convenient layout. To this end, the header file is self-explanatory.

The wsdl2h tool generates only one file, the header file that includes all of the information obtained from all WSDL and schema files provided to the tool at the command-line prompt. The default output file name of wsdl2h is the first WSDL/schema input file name but with extension .h instead of .wsdl (or .xsd). When an input file is absent or a WSDL file from a Web location is accessed, the header output will be produced on the standard output unless option -o is used to direct the output to a file.

The wsdl2h command-line options are:

Option	Description
-a	generate indexed struct names for local elements with anonymous types
-b	bi-directional operations to serve one-way response messages (duplex)
-c	generate C source code
-d	use DOM to populate xs:any and xsd:anyType elements
-e	don't qualify enum names
	This option is for backward compatibility with gSOAP 2.4.1 and earlier.
	The option does not produce code that conforms to WS-I Basic Profile 1.0a.
-f	generate flat C++ class hierarchy for schema extensions
-g	generate global top-level element declarations
-h	print help information
-I path	use path to locate source files for #import
-i	don't import (advanced option)
-j	don't generate SOAP_ENV_ _Header and SOAP_ENV_ _Detail definitions
-k	don't generate SOAP_ENV_ _Header mustUnderstand qualifiers
-l	include license information in output
-m	use xsd.h module to import primitive types
-N name	use name for service prefixes to produce a service for each binding
-n name	use name as the base namespace prefix name instead of ns
-o file	output to file
-P	don't create polymorphic types inherited from xsd_ _anyType
-p	create polymorphic types inherited from base xsd_ _anyType
	This is automatically performed when WSDL contains polymorphic definitions
-q name	use name for the C++ namespace of all declarations
-r host[:port[:uid:pwd]]	connect via proxy host, port, and proxy credentials
-r:uid:pwd	connect with authentication credentials (digest auth requires SSL)
-R	generate REST operations for REST bindings in the WSDL
-s	don't generate STL code (no std::string and no std::vector)
-t file	use type map file instead of the default file typemap.dat
-u	don't generate unions
-v	verbose output
-W	suppress warnings
-w	always wrap response parameters in a response struct
-x	don't generate _XML any/anyAttribute extensibility elements
-y	generate typedef synonyms for structs and enums
-z1	compatibility with 2.7.6e: generate pointer-based arrays
-z2	compatibility with 2.7.15: qualify element/attribute referenced members
-z3	compatibility with 2.7.16 to 2.8.7: qualify element/attribute references
-z4	compatibility up to 2.8.11: don't generate union structs in std::vector
-z5	compatibility up to 2.8.15
-z5	compatibility up to 2.8.17
-_	don't generate _USCORE (replace with UNICODE _x005f)

Note: see README.txt in the wsdl directory for the latest information on installation and options to of the wsdl2h WSDL/schema importer.

8.2   Customizing Data Bindings With The typemap.dat File

The typemap.dat file for the wsdl2h tool is intended to customize or optimize the type bindings by mapping schema types to C/C++ types. It contains custom XML Schema to C/C++ type bindings

and a few bindings are defined for convenience.

Here is an example typemap file's content:

```
# This file contains custom definitions of the XML Schema types and
# C/C++ types for your project, and XML namespace prefix definitions.
# The wsdl2h WSDL importer consults this file to determine bindings.

[
// This comment will be included in the generated .h file
// You can include any additional declarations, includes, imports, etc.
// within [ ] sections.  The brackets MUST appear at the start of a line
]
# XML namespace prefix definitions can be provided to override the
# default choice of ns1, ns2, ...  prefixes.  For example:

i = "http://www.soapinterop.org/"
s = "http://www.soapinterop.org/xsd"
```

Type bindings can be provided to bind XML schema types to C/C++ types for your project. Type bindings have four parts:

```
prefix__type = declaration | use | ptr-use
```

where 'prefix__type' is the C/C++-translation of the schema type, 'declaration' introduces the C/C++ type in the header file, the optional 'use' specifies how the type is used directly, and the optional 'ptr-use' specifies how the type is used as a pointer type.

```
# Example XML Schema and C/C++ type bindings:

xsd__int = | int
xsd__string = | char* | char*
xsd__boolean = enum xsd__boolean  false_, true_ ; | enum xsd__boolean
xsd__base64Binary = class xsd__base64Binary  unsigned char *__ptr; int __size; ; |
xsd__base64Binary | xsd__base64Binary
# You can extend structs and classes with member data and functions.
# For example, adding a constructor to ns__myClass:  ns__myClass = $ ns__myClass();
# The general form is # class_name = $ member;
```

The i and s prefixes are declared such that the header file output by the WSDL parser will use these to produce C/C++ code. XML Schema types are associated with an optional C/C++ type declaration, a use reference, and a pointer-use reference. The pointer-use reference of the xsd__byte type for example, is **int*** because **char*** is reserved for strings.

When a type binding requires only the usage to be changed, the declaration part can be given by an elipsis ..., as in:

```
prefix__type = ...  | use | ptr-use
```

This ensures that the wsdl2h-generated type definition is preserved, while the use and ptr-use are remapped.

This method is useful to serialize dynamic types in C, where elements types int XML carry the `xsi:type` attribute.

The following example illustrates an "any" type mapping for the `ns:sometype` XSD type in a schema. This type will be replaced with a "any" type wrapper that supports dynamic serialization with `xsi:type`:

```
[
struct __any
{
  int __type;
  void *__item;
}
]
xsd__anyType = ...  | struct __any | struct __any
```

where `__type` and `__item` are used to (de)serialize any data type in the wrapper, including base and its derived types based on `xsi:type` attribuation.

To support complexType extensions that are dynamically bound in C code, i.e. polymorphic types based on inheritance hierarchies, we can redeclare the base type of a hierarchy as a wrapper type and use the `__type` to serialize base or derived types. One addition is needed to support base type serialization without the use of `xsi:type` attributes. The absence of this attribute requires the serialization of the base type.

Basically, we need to be able to both handle a base type and its extensions as per schema extensibility. Say base type `ns:base` is a complexType that is extended by several other complexTypes. To implement dynamic binding in C to serialize the base type and derived types, we define:

```
[
struct __ns__base
{
  int __type;
  void *__item;
  struct ns__base *__self;
}
]
ns__base = ...  | struct __ns__base | struct __ns__base
```

The `__self` field refers to the element tag (basically a struct member name) to which the `ns:base` type is associated. So for example, we see in the soapcpp2-generated output:

```
struct ns__data
{
  ...
  struct __ns__base name;
  ...
};
```

89

where `__item` represents `name` when the `__ns__base` is serialized with an `xsi:type` attribute, and `__self` represents `name` when the `__ns__base` is serialized wwithout an `xsi:type` attribute. Therefore, the dynamic binding defaults to **struct** `ns__base *__self` when no dynamic type information in XML is available.

Additional data and function members can be provided to extend a generated struct or class. Class and struct extensions are of the form:

```
prefix__type = $ member-declaration
```

For example, to add a constructor and destructor to class myns__record:

```
myns__record = $ myns__record(); myns__record = $ ~myns__record();
```

Type remappings can be given to map a type to another type:

```
prefix__type1 == prefix__type2
```

which replaces `prefix__type1` by `prefix__type2` in the wsdl2h output. For example:

```
SOAP_ENC__boolean == xsd__boolean
```

where `SOAP_ENC__boolean` is mapped to `xsd__boolean`, which in turn may be mapped to a C `enum` `xsd__boolean` type or C++ `bool` type.

9   Using the soapcpp2 Compiler and Code Generator

The `soapcpp2` compiler and code generator is invoked from the command line and optionally takes the name of a header file as an argument or, when the file name is absent, parses the standard input:

> `> soapcpp2 [aheaderfile.h]`

where `aheaderfile.h` is a C/C++ header file generated by `wsdl2h` or developed manually to specify the SOAP/XML service operations as function prototypes and the C/C++ data types to be auto-mapped to XML.

The `soapcpp2` tool produces C/C++ source files. These files are used to implement SOAP/XML clients and services, and to implement the advanced XML data binding logic to convert C/C++ data into XML and vice versa.

The type of files generated by `soapcpp2` are:

File Name	Description
soapStub.h	A modified and annotated header file produced from the input header file
soapH.h	Main header file to be included by all client and service sources
soapC.cpp	Serializers and deserializers for the specified data structures
soapClient.cpp	Client stub routines for remote operations
soapServer.cpp	Service skeleton routines
soapClientLib.cpp	Client stubs combined with local static (de)serializers
soapServerLib.cpp	Service skeletons combined with local static (de)serializers
soapXYZProxy.h	A C++ proxy object (link with soapC.cpp soapClient.cpp)
soapXYZProxy.h	With option -i: proxy object (link with soapC.cpp and soapXYZProxy.cpp)
soapXYZProxy.cpp	With option -i: proxy code
soapXYZObject.h	A C++ server object (link with soapC.cpp and soapServer.cpp)
soapXYZService.h	With option -i: server object (link with soapC.cpp and soapXYZService.cpp)
soapXYZService.cpp	With option -i: server code
.xsd	An ns.xsd file is generated with an XML Schema for each namespace prefix ns used by a data structure in the header file input to the compiler, see Section 7.2.9
.wsdl	A ns.wsdl file is generated with an WSDL description for each namespace prefix ns used by a service operation in the header file input to the compiler, see Section 7.2.9
.xml	Several SOAP/XML request and response files are generated. These are example message files are valid provided that sufficient schema namespace directives are added to the header file or the generated .nsmap namespace table for the client/service is not modified by hand
.nsmap	A ns.nsmap file is generated for each namespace prefix ns used by a service operation in the header file input to the compiler, see Section 7.2.9. The file contains a namespace mapping table that can be used in the client/service sources

Both client and service applications are developed from a header file that specifies the service operations. If client and service applications are developed with the same header file, the applications are guaranteed to be compatible because the stub and skeleton routines use the same serializers and deserializers to encode and decode the parameters. Note that when client and service applications are developed together, an application developer does not need to know the details of the internal SOAP encoding used by the client and service.

The soapClientLib.cpp and soapServerLib.cpp can be used to build (dynamic) client and server libraries. The serialization routines are local (static) to avoid link symbol conflicts. You must create a separate library for SOAP Header and Fault handling, as described in Section 19.36.

The following files are part of the gSOAP package and are required to build client and service applications:

File Name	Description
stdsoap2.h	Header file of stdsoap2.cpp runtime library
stdsoap2.c	Runtime C library with XML parser and run-time support routines
stdsoap2.cpp	Runtime C++ library identical to stdsoap2.c

9.1 soapcpp2 Options

The soapcpp2 source-to-source compiler supports the following command-line options:

Option	Description
-1	Generate SOAP 1.1 bindings
-2	Generate SOAP 1.2 bindings
-0	No SOAP bindings, use REST
-C	Generate client-side code only
-S	Generate server-side code only
-T	Generate server auto-test code
-L	Do not generate soapClientLib/soapServerLib
-a	Use SOAPAction with WS-Addressing to invoke server-side operations
-A	Require SOAPAction to invoke server-side operations
-b	serialize byte arrays char[N] as string
-c	Generate pure C code
-d <path>	Save sources in directory specified by <path>
-e	Generate SOAP RPC encoding style bindings
-f N	File split of N XML serializer implementations per file
-h	Print a brief usage message
-i	Generate service proxies and objects inherited from soap struct
-j	Generate C++ service proxies and objects that can share a soap struct
-I <path>	Use <path> for #import (paths separated with ':' or ';' for windows)
-k	generate data structure walkers (experimental)
-l	Generate linkable modules (experimental)
-m	Generate Matlab[tm] code for MEX compiler
-n	When used with -p, enables multi-client and multi-server builds:
	Sets compiler option WITH_NONAMESPACES, see Section 9.11
	Saves the namespace mapping table with name <name>_namespaces instead of namespaces
	Renames soap_serve() into <name>_serve() and soap_destroy() into <name>_destroy()
-p <name>	Save sources with file name prefix <name> instead of "soap"
-q <name>	Use name for the C++ namespace of all declarations
-s	Generates deserialization code with strict XML validation checks
-t	Generates code to send typed messages (with the xsi:type attribute)
-u	uncomment comments in WSDL/schema output by suppressing XML comments
-v	Display version info
-w	Do not generate WSDL and schema files
-x	Do not generate sample XML message files
-y	include C/C++ type access information in sample XML messages
-z1	compatibility: generate old-style C++ service proxies and objects
-z2	compatibility with 2.7.x: omit XML output for NULL pointers

For example

> soapcpp2 -cd '../projects' -pmy file.h

Saves the sources:

../projects/myH.h
../projects/myC.c
../projects/myClient.c
../projects/myServer.c
../projects/myStub.h

MS Windows users can use the usual "/" for options, for example:

```
soapcpp2 /cd '..\projects' /pmy file.h
```

Compiler options c, i, n, l, w can be set in the gSOAP header file using the //gsoapopt directive. For example,

```
// Generate pure C and do not produce WSDL output:
//gsoapopt cw
int ns__myMethod(char*,char**); // takes a string and returns a string
```

9.2 SOAP 1.1 Versus SOAP 1.2 and Dynamic Switching

gSOAP supports SOAP 1.1 by default. SOAP 1.2 support is automatically turned on when the appropriate SOAP 1.2 namespace is used, which shows up in the namespace mapping table:

```
struct Namespace namespaces[] =
{
   {"SOAP-ENV", "http://www.w3.org/2003/05/soap-envelope", ... },
   {"SOAP-ENC", "http://www.w3.org/2003/05/soap-encoding, ... "},
   ...
}
```

Normally the soapcpp2-generated namespace table allows dynamic switching between SOAP 1.1 to SOAP 1.2 by providing the SOAP 1.2 namespace as a **pattern** in the third column of a namespace table:

```
struct Namespace namespaces[] =
{
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/", "http://www.w3.org/*/soap-encoding"},
   {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/", "http://www.w3.org/*/soap-envelope"},
   ...
}
```

where the "*" in the third column of the namespace URI pattern is a meta wildcard. This is used to match and accept inbound namespaces.

This way, gSOAP Web services can respond to either SOAP 1.1 or SOAP 1.2 requests. gSOAP will automatically return SOAP 1.2 responses for SOAP 1.2 requests.

The gSOAP soapcpp2 tool generates a .nsmap file with SOAP-ENV and SOAP-ENC namespace patterns similar to the above. Since clients issue a send first, they will always use SOAP 1.1 for requests when the namespace table is similar as shown above. Clients can accept SOAP 1.2 responses by inspecting the response message.

To use SOAP 1.2 by default and allow SOAP 1.1 messages to be received, use the soapcpp2 -2 option to generate SOAP 1.2 conformant .nsmap and .wsdl files. Alternatively, add the following line to your service definitions header file (generated by wsdl2h) for soapcpp2:

```
#import "import/soap12.h"
```

Caution: SOAP 1.2 does not support partially transmitted arrays. So the __offset field of a dynamic array is meaningless.

Caution: SOAP 1.2 requires the use of SOAP_ENV__Code, SOAP_ENV__Reason, and SOAP_ENV__Detail fields in a SOAP_ENV__Fault fault struct, while SOAP 1.1 uses faultcode, faultstring, and detail fields. Use soap_receiver_fault_subcode(**struct** soap *soap, **const char** *subcode, **const char** *faultstring, **const char** *detail) to set a SOAP 1.1/1.2 fault at the server-side with a fault subcode (SOAP 1.2). Use soap_sender_fault_subcode(**struct** soap *soap, **const char** *subcode, **const char** *faultstring, **const char** *detail) to set a SOAP 1.1/1.2 unrecoverable Bad Request fault at the server-side with a fault subcode (SOAP 1.2).

9.3 The soapdefs.h Header File

The soapdefs.h header file is included in stdsoap2.h when compiling with option -DWITH_SOAPDEFS_H:

```
> c++ -DWITH_SOAPDEFS_H -c stdsoap2.cpp
```

The soapdefs.h file allows users to include definitions and add includes without requiring changes to stdsoap2.h. For example,

```
// Contents of soapdefs.h
#include <ostream>
#define SOAP_BUFLEN 65536 // use large send/recv buffer
```

The following header file can now refer to ostream:

```
extern class ostream; // ostream can't be (de)serialized, but need to be declared to make it visible
to gSOAP
class ns__myClass
{ ...
   virtual void print(ostream &s) const; // need ostream here
   ...
};
```

See also Section 19.3.

9.4 How to Build Modules and Libraries with the #module Directive

The #module directive is used to build modules. A library can be build from a module and linked with multiple Web services applications. The directive should appear at the top of the header file and has the following formats:

```
#module "name"
```

and

94

```
#module "name" "fullname"
```

where *name* must be a unique short name for the module. The name is case insensitive and MUST not exceed 4 characters in length. The *fullname*, when present, represents the full name of the module.

The rest of the content of the header file includes type declarations and optionally the declarations of service operations and SOAP Headers/Faults. When the gSOAP soapcpp2 compiler processes the header file module, it will generate the source codes for a library. The Web services application that uses the library should use a header file that imports the module with the #import directive.

For example:

```
/* Contents of module.h */
#module "test"
long;
char*;
struct ns__S
{ ... }
```

The module.h header file declares a **long**, **char***, and a **struct** ns__X. The module name is "test", so the gSOAP soapcpp2 compiler produces a testC.cpp file with the (de)serializers for these types. The testC.cpp library can be separately compiled and linked with an application that is built from a header file that imports "module.h" using #import "module.h". You should also compile testClient.cpp when you want to build a library that includes the service opertions that you defined in the module header file.

There are some limitations on a sequence of module imports. A module MUST be imported into another header to use the module content and you MUST place this import statement before all other statements in the file, including other imports (except when these are also modules). It is also advised to put all basic data type definitions in the root module of a module import hierarchy, e.g. using **typedef** to declare XSD types (see also Section 11.3).

You cannot use a module alone to build a SOAP or XML application. That is, the final gSOAP header file in the import chain SHOULD NOT be a module.

When multiple modules are linked, the types that they declare MUST be declared in one module only to avoid name clashes and link errors. You cannot create two modules that share the same type declaration and link the modules. When necessary, you should consider creating a module hierarchy such that types are declared only once and by only one module when these modules must be linked.

9.5   How to use the #import Directive

The #import directive is used to include gSOAP header files into other gSOAP header files for processing with the gSOAP compiler soapcpp2. The C #include directive cannot be used to include gSOAP header files. The #include directive is reserved to control the post-gSOAP compilation process, see 9.6.

The #import directive is used for two purposes: you can use it to include the contents of a header file into another header file and you can use it to import a module, see 9.4.

An example of the #import directive:

```
#import "mydefs.gsoap"
int ns__mymethod(xsd__string in, xsd__int *out);
```

where "mydefs.gsoap" is a gSOAP header file that defines xsd__string and xsd__int:

```
typedef char *xsd__string;
typedef int xsd__int;
```

When importing a module, where the module content is declared with #module, then note that this module MUST place the import statement before all other statements in the header file, including other imports (except when these are also modules).

9.6 How to Use #include and #define Directives

The #include and #define directives are normally ignored by the gSOAP soapcpp2 compiler and just passed on to the generated code. Thus, the gSOAP compiler will not actually parse the contents of the header files provided by the #include directives in a header file. Instead, the #include and #define directives will be added to the generated soapH.h header file **before** any other header file is included. Therefore, #include and #define directives can be used to control the C/C++ compilation process of the sources of an application. However, they have no effect on soapcpp2.

The following example header file refers to ostream by including <ostream>:

```
#include <ostream>
#define WITH_COOKIES // use HTTP cookie support (you must compile stdsoap2.cpp with -
DWITH_COOKIES)
#define WITH_OPENSSL // enable HTTPS/SSL support (you must compile stdsoap2.cpp with
-DWITH_OPENSSL)
#define WITH_GNUTLS // enable HTTPS/SSL support (you must compile stdsoap2.cpp with -
DWITH_GNUTLS)
#define SOAP_DEFAULT_float FLT_NAN // use NaN instead of 0.0
extern class ostream; // ostream can't be (de)serialized, but need to be declared to make it visible
to gSOAP
class ns__myClass
{ ...
   virtual void print(ostream &s) const; // need ostream here
   ...
};
```

This example also uses #define directives for various settings in the target source code.

Caution: Note that the use of #define in the header file does not automatically result in compiling stdsoap2.cpp with these directives. You MUST use the -DWITH_COOKIES and -DWITH_OPENSSL (or -DWITH_GNUTLS options when compiling stdsoap2.cpp before linking the object file with your codes. As an alternative, you can use #define WITH_SOAPDEFS_H and put the #define directives in the soapdefs.h file.

9.7 Compiling a SOAP/XML Client Application with soapcpp2

After invoking the gSOAP soapcpp2 tool on a header file description of a service, the client application can be compiled on a Linux machine as follows:

```
> c++ -o myclient myclient.cpp stdsoap2.cpp soapC.cpp soapClient.cpp
```

Or on a Unix machine:

```
> c++ -o myclient myclient.cpp stdsoap2.cpp soapC.cpp soapClient.cpp -lsocket -lxnet -lnsl
```

(Depending on your system configuration, the libraries libsocket.a, libxnet.a, libnsl.a or dynamic *.so versions of those libraries are required.)

The myclient.cpp file must include soapH.h and must define a global namespace mapping table. A typical client program layout with namespace mapping table is shown below:

```
// Contents of file "myclient.cpp"
#include "soapH.h";
...
// A service operation invocation:
   soap_call_some_remote_method(...);
...
struct Namespace namespaces[] =
{   // {"ns-prefix", "ns-name"}
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
   {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
   {"xsi", "http://www.w3.org/2001/XMLSchema-instance"},
   {"xsd", "http://www.w3.org/2001/XMLSchema"},
   {"ns1", "urn:my-remote-method"},
   {NULL, NULL}
};
...
```

A mapping table is generated by the gSOAP soapcpp2 compiler that can be used in the source, see Section 7.2.9.

9.8 Compiling a SOAP/XML Web Service with soapcpp2

After invoking the gSOAP soapcpp2 tool on a header file description of the service, the server application can be compiled on a Linux machine as follows:

```
> c++ -o myserver myserver.cpp stdsoap2.cpp soapC.cpp soapServer.cpp
```

Or on a Unix machine:

```
> c++ -o myserver myserver.cpp stdsoap2.cpp soapC.cpp soapServer.cpp -lsocket -lxnet -lnsl
```

(Depending on your system configuration, the libraries libsocket.a, libxnet.a, libnsl.a or dynamic *.so versions of those libraries are required.)

The myserver.cpp file must include soapH.h and must define a global namespace mapping table. A typical service program layout with namespace mapping table is shown below:

```
// Contents of file "myserver.cpp"
#include "soapH.h";
int main()
{
   soap_serve(soap_new());
}
...
// Implementations of the service operations as C++ functions
...
struct Namespace namespaces[] =
{   // {"ns-prefix", "ns-name"}
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
   {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
   {"xsi", "http://www.w3.org/2001/XMLSchema-instance"},
   {"xsd", "http://www.w3.org/2001/XMLSchema"},
   {"ns1", "urn:my-remote-method"},
   {NULL, NULL}
};
...
```

When the gSOAP service is compiled and installed as a CGI application, the soap_serve function acts as a service dispatcher. It listens to standard input and invokes the method via a skeleton routine to serve a SOAP client request. After the request is served, the response is encoded in SOAP and send to standard output. The method must be implemented in the server application and the type signature of the method must be identical to the service operations specified in the header file. That is, the function prototype in the header file must be a valid prototype of the method implemented as a C/C++ function.

9.9   Compiling Web Services and Clients in ANSI C

The gSOAP soapcpp2 compiler can be used to create pure C Web services and clients. The gSOAP stub and skeleton compiler soapcpp2 generates .cpp files by default. The compiler generates .c files with the -c option. However, these files only use C syntax and data types **if** the header file input to soapcpp2 uses C syntax and data types. For example:

```
> soapcpp2 -c quote.h
> cc -o quote quote.c stdsoap2.c soapC.c soapClient.c
```

Warnings will be issued by the compiler when C++ class declarations occur in the header file.

9.10   Limitations of gSOAP

gSOAP is SOAP 1.1 and SOAP 1.2 compliant and supports SOAP RPC and document/literal operations.

From the perspective of the C/C++ language, a few C++ language features are not supported by gSOAP and these features cannot be used in the specification of SOAP service operations.

There are certain limitations for the following C++ language constructs:

STL and STL templates The gSOAP `soapcpp2` compiler supports C++ strings `std::string` and `std::wstring` (see Section 11.3.6) and the STL containers `std::deque`, `std::list`, `std::vector`, and `std::set`, (see Section 11.11.8).

Templates The gSOAP `soapcpp2` compiler is a preprocessor that cannot determine the template instantiations used by the main program, nor can it generate templated code. You can however implement containers similar to the STL containers.

Multiple inheritance Single class inheritance is supported. Multiple inheritance cannot be supported due to limitations of the SOAP protocol.

Abstract methods A class must be instantiatable to allow decoding of instances of the class.

Directives Directives and pragmas such as #include and #define are interpreted by the gSOAP `soapcpp2` compiler. However, the interpretation is different compared to the usual handling of directives, see Section 9.6. If necessary, a traditional C++ preprocessor can be used for the interpretation of directives. For example, Unix and Linux users can use "`cpp -B`" to expand the header file, e.g. `cpp -B myfile.h | soapcpp2`. Use the gSOAP #import directive to import gSOAP header files, see 9.5.

C and C++ programming statements All class methods of a class should be declared within the class declaration in the header file, but the methods should not be implemented in code. All class method implementations must be defined within another C++ source file and linked to the application.

The following data types require some attention to ensure they are serialized:

union types A **union** data type can not be serialized unless run-time information is associated with a **union** in a struct/class as discussed in Section 11.7. An alternative is to use a **struct** with a pointer type for each field. Because `NULL` pointers are not encoded, the resulting encoding will appear as a union type if only one pointer field is valid (i.e. non-`NULL`) at the time that the data type is encoded.

void and void* types The **void** data type cannot be serialized unless run-time type information is associated with the pointer using a **int** __type field in the struct/class that contains the **void***. The **void*** data type is typically used to point to some object or to some array of some type of objects at run-time. The compiler cannot determine the type of data pointed to and the size of the array pointed to. A struct or class with a **void*** field can be augmented to support the (de)serialization of the **void*** using a **int** __type field as described in Section 11.9.

Pointers to sequences of elements in memory Any pointer, except for C strings which are pointers to a sequence of characters, are treated by the compiler as if the pointer points to **only one element in memory** at run-time. Consequently, the encoding and decoding routines will ignore any subsequent elements that follow the first in memory. For the same reason, arrays of undetermined length, e.g. **float** a[] cannot be used. gSOAP supports dynamic arrays using a special type convention, see Section 11.11.

Uninitialized pointers Obviously, all pointers that are part of a data structure must be valid or NULL to enable serialization of the data structure at run time.

There are a number of programming solutions that can be adopted to circumvent these limitations. Instead of using **void***, a program can in some cases be modified to use a pointer to a known type. If the pointer is intended to point to different types of objects, a generic base class can be declared and the pointer is declared to point to the base class. All the other types are declared to be derived classes of this base class. For pointers that point to a sequence of elements in memory dynamic arrays should be used instead, see 11.11.

9.11 Library Build Flags

The following macros (#defines) can be used to enable certain optional features when building the libgsoap library or when compiling and linking stdsoap2.c and stdsoap2.cpp:

Macro	Description
WITH_SOAPDEFS_H	includes the **soapdefs.h** file for custom settings, see Section 9.3
WITH_COOKIES	enables HTTP cookies, see Sections 19.28 19.29
WITH_OPENSSL	enables OpenSSL, see Sections 19.22 19.21
WITH_GNUTLS	enables GNUTLS, see Sections 19.22 19.21
WITH_IPV6	enables IPv6 support (compile ALL sources with this macro set)
WITH_IPV6_V6ONLY	IPv6-only server option (compile ALL sources with this macro set)
WITH_NO_IPV6_V6ONLY	resets IPv6-only server option (compile ALL sources with this macro set)
WITH_TCPFIN	use TCP FIN after sends when socket is ready to close
WITH_FASTCGI	enables FastCGI, see Sections 19.31
WITH_GZIP	enables gzip and deflate compression, see Section 19.27
WITH_ZLIB	enables deflate compression only, see Section 19.27
WITH_FAST	use faster memory allocation for WITH_LEAN/WITH_LEANER
WITH_NOIO	eliminates need for file IO and BSD socket library, see Section 19.33
WITH_NOIDREF	eliminates href/ref and id attributes to (de)serialize multi-ref data, or alternatively use the SOAP_XML_TREE runtime flag
WITH_NOHTTP	eliminates HTTP stack to reduce code size
WITH_NOZONE	silently ignores the timezone in time conversions
WITH_LEAN	creates a small-footprint executable, see Section 19.32
WITH_LEANER	creates an even smaller footprint executable, see Section 19.32
WITH_COMPAT	removes dependency on C++ stream libraries, eliminating C++ exceptions
WITH_NONAMESPACES	removes dependence on global **namespaces** table, MUST set it explicitly with **soap_set_namespaces()** see also Section 10.4
WITH_PURE_VIRTUAL	for C++ abstract service classes with pure virtual methods
WITH_NOEMPTYSTRUCT	inserts a dummy member in empty structs to allow compilation
WITH_NOGLOBAL	omit SOAP Header and Fault serialization code
WITH_NOCDATA	do not retain the parsed CDATA sections in literal XML strings (convert)
WITH_CDATA	retain the parsed CDATA sections in literal XML strings (no conversion, default)
WITH_C_LOCALE	use locale functions when available to ensure locale-independent number conversions (force the use of C locale)
WITH_CASEINSENSITIVETAGS	enable case insensitive XML parsing
WITH_REPLACE_ILLEGAL_UTF8	replaces UTF8 content that is outside the allowed range of XML 1.0

Other compile-time flags:

Macro	Description
SOCKET_CLOSE_ON_EXIT	prevents a server port from staying in listening mode after exit by internally setting **fcntl(sock, F_SETFD, FD_CLOEXEC)**

Compile-time flags to change the default engine settings:

Macro	Description
SOAP_BUFLEN	the length of the internal message buffer (affects socket comms)
SOAP_TAGLEN	maximum length of XML tags and URL domain names (buffering)
SOAP_SSL_RSA_BITS	the length of the RSA key (2048 by default)
SOAP_UNKNOWN_CHAR	an 8 bit code that represents a character that could not be converted to an ASCII **char** (e.g. from Unicode, applicable when SOAP_C_UTFSTRING is off)

Caution: it is important that all of these macros MUST be consistently defined to compile all sources, such as stdsoap2.cpp, soapC.cpp, soapClient.cpp, soapServer.cpp, and all application sources that

include stdsoap2.h or soapH.h. If the macros are not consistently used, the application will crash due to a mismatches in the declaration and access of the gSOAP context.

9.12   Run Time Flags

gSOAP provides flags to control the input and output mode settings at runtime. These flags are divided into four categories: transport (IO), content encoding (ENC), XML marshalling (XML), and C/C++ data mapping (C).

Although gSOAP is fully SOAP 1.1 compliant, some SOAP implementations may have trouble accepting multi-reference data and/or require explicit nil data so these flags can be used to put gSOAP in "safe mode". In addition, the embedding (or inlining) of multi-reference data is adopted in the SOAP 1.2 specification, which gSOAP automatically supports when handling with SOAP 1.2 messages.

To set and clear flags for inbound message processing use:

```
soap_set_imode(soap, inflag);
soap_clr_imode(soap, inflag);
```

To set and clear the flags for outbound message processing use:

```
soap_set_omode(soap, outflag);
soap_clr_imode(soap, outflag);
```

To allocate and initialize a gSOAP context with inbound and outbound flags use:

```
soap_new2(soap, inflag, outflag);
```

To initialize an unitialized gSOAP context with inbound and outbound flags use:

```
soap_init2(soap, inflag, outflag);
```

The input-mode and output-mode flags for inbound and outbound message processing are:

Flag	Description
SOAP_IO_FLUSH	in: disable buffering and flush output (default for all file-based output)
SOAP_IO_BUFFER	in: enable buffering (default for all socket-oriented connections)
SOAP_IO_STORE	in: store entire message to calculate HTTP content length
SOAP_IO_CHUNK	out: use HTTP chunking
SOAP_IO_LENGTH	out: (internal flag) require apriori calculation of content length
SOAP_IO_KEEPALIVE	in&out: attempt to keep socket connections alive (open)
SOAP_IO_UDP	in&out: use UDP (datagram) transport, maximum message length is SOAP_BUFLEN
SOAP_ENC_XML	out: use plain XML encoding without HTTP headers (useful with SOAP_ENC_ZLIB)
SOAP_ENC_DIME	out: use DIME encoding (automatic when DIME attachments are used)
SOAP_ENC_MIME	out: use MIME encoding (activate using soap_set_mime)
SOAP_ENC_MTOM	out: use MTOM XOP attachments (instead of DIME)
SOAP_ENC_ZLIB	out: compress encoding with Zlib (deflate or gzip format)
SOAP_ENC_SSL	in&out: encrypt with SSL (automatic with "https:" endpoints)
SOAP_XML_INDENT	out: produces indented XML output
SOAP_XML_CANONICAL	out: produces canonical XML output
SOAP_XML_DEFAULTNS	out: produces xmlns="..." default binding namespaced output
SOAP_XML_IGNORENS	in: ignores the use of XML namespaces in input
SOAP_XML_STRICT	in: XML strict validation
SOAP_XML_TREE	out: serialize data as XML trees (no multi-ref, duplicate data when necessary)
	in: ignore id attributes (do not resolve id-ref)
SOAP_XML_GRAPH	out: serialize data as an XML graph with inline multi-ref (SOAP 1.2 default)
SOAP_XML_NIL	out: serialize NULL data as xsi:nil attributed elements
SOAP_XML_NOTYPE	out: disable xsi:type attributes
SOAP_C_NOIOB	in: do not fault with SOAP_IOB
SOAP_C_UTFSTRING	in&out: (de)serialize 8-bit strings "as is" (strings MUST have UTF-8 encoded content)
SOAP_C_MBSTRING	in&out: enable multibyte character support (depends on locale)
SOAP_C_NILSTRING	out: serialize empty strings as nil (ommited element)

The flags can be selectively turned on/off at any time, for example when multiple Web services are accessed by a client that require special treatment.

All flags are orthogonal, except SOAP_IO_FLUSH, SOAP_IO_BUFFER, SOAP_IO_STORE, and SOAP_IO_CHUNK which are enumerations and only one of these I/O flags can be used. Also the XML serialization flags SOAP_XML_TREE and SOAP_XML_GRAPH should not be mixed.

The flags control the inbound and outbound message transport, encoding, and (de)serialization. The following functions are used to set and reset the flags for input and output modes:

Function	Description
soap_init2(**struct** soap *soap, int imode, int omode)	Initialize the runtime and set flags
soap_imode(**struct** soap *soap, int imode)	Set all input mode flags
soap_omode(**struct** soap *soap, int omode)	Set all output mode flags
soap_set_imode(**struct** soap *soap, int imode)	Enable input mode flags
soap_set_omode(**struct** soap *soap, int omode)	Enable output mode flags
soap_clr_imode(**struct** soap *soap, int omode)	Disable input mode flags
soap_clr_omode(**struct** soap *soap, int omode)	Disable output mode flags

The default setting is SOAP_IO_DEFAULT for both input and output modes.

For example

```
struct soap soap;
```

```
soap_init2(&soap, SOAP_IO_KEEPALIVE,
    SOAP_IO_KEEPALIVE|SOAP_ENC_ZLIB|SOAP_XML_TREE|SOAP_XML_CANONICAL);
if (soap_call_ns__myMethod(&soap, ...))
...
```

sends a compressed client request with keep-alive enabled and all data serialized as canonical XML trees.

In many cases, setting the input mode will have no effect, especially with HTTP transport because gSOAP will determine the optimal input buffering and the encoding used for an inbound message. The flags that have an effect on handling inbound messages are SOAP_IO_KEEPALIVE, SOAP_ENC_SSL (but automatic when "https:" endpoints are used or soap_ssl_accept), SOAP_C_NOIOB, SOAP_C_UTFSTRING, and SOAP_C_MBSTRING.

Caution: The SOAP_XML_TREE serialization flag can be used to improve interoperability with SOAP implementations that are not fully SOAP 1.1 compliant. However, a tree serialization will duplicate data when necessary and will crash the serializer for cyclic data structures.

Additional run-time flags to control sockets.

Use the following selection of flags that are OS dependent to control sockets for send/sendto/recv/recvfrom operations:

socket_flags	Description
MSG_NOSIGNAL	disables sigpipe (check your OS, this is not portable)
MSG_DONTROUTE	bypass routing, use direct interface

Use the following selection of flags to set client-side socket connection flags (setsockopt):

connect_flags	Description
SO_NOSIGPIPE	disables sigpipe (check your OS, this is not portable)
SO_DEBUG	turns on recording of debugging information in the underlying protocol modules
SO_BROADCAST	permits sending of broadcast messages (e.g. with UDP) when permitted
SO_LINGER	set soap.linger_time (set this value as needed)

Use the following selection of flags to set server-side socket connection accept flags (setsockopt):

accept_flags	Description
SO_NOSIGPIPE	disables sigpipe (check your OS, this is not portable)
SO_DEBUG	turns on recording of debugging information in the underlying protocol modules
SO_REUSEADDR	reuse bind address immediately (prevents bind reject)
SO_LINGER	set soap.linger_time (set this value as needed)

For example, soap.accept_flags = (SO_NOSIGPIPE — SO_LINGER) disables sigpipe signals and set linger time value given by soap.linger_time (zero by default).

9.13 Memory Management

Understanding gSOAP's run-time memory management is important to optimize client and service applications by eliminating memory leaks and/or dangling references.

There are two forms of dynamic (heap) allocations made by gSOAP's runtime for serialization and deserialization of data. Temporary data is created by the runtime such as hash tables to keep pointer reference information for serialization and hash tables to keep XML id/href information for multi-reference object deserialization. Deserialized data is created upon receiving SOAP messages. This data is stored on the heap and requires several calls to the malloc library function to allocate space for the data and **new** to create class instances. All such allocations are tracked by gSOAP's runtime by linked lists for later deallocation. The linked list for malloc allocations uses some extra space in each malloced block to form a chain of pointers through the malloced blocks. A separate malloced linked list is used to keep track of class instance allocations.

If you want to preserve the deserialized data before deleting a soap context, you can assign management of the data and delegate responsibility of deletion to another soap context using soap_delegate_deletion(**struct** soap *soap_from, **struct** soap *soap_to). This moves all deserialized and temporary data to the other soap context soap_to, which will delete its data and all the delegated data it is responsible for when you call soap_destroy and soap_end. This can be particularly useful for making client calls inside a server operation, i.e. a mixed server/client. The client call inside the server operation requires a new soap context, e.g. copied from the server's with soap_copy. Before destroying the client context with soap_free, the data can be delegated to the server's context with soap_delegate_deletion. See samples/mashup/machupserver.c code for an example.

Note that gSOAP does not per se enforce a deallocation policy and the user can adopt a deallocation policy that works best for a particular application. As a consequence, deserialized data is never deallocated by the gSOAP runtime unless the user explicitly forces deallocation by calling functions to deallocate data collectively or individually.

The deallocation functions are:

Function Call	Description
soap_destroy(**struct** soap *soap)	Remove all dynamically allocated C++ objects. must be called before soap_end()
soap_end(**struct** soap *soap)	Remove temporary data and deserialized data except class instances
soap_free_temp(**struct** soap *soap)	Instead of soap_destroy and soap_end: remove temporary data only
soap_dealloc(**struct** soap *soap, **void** *p)	Remove malloced data at p. When p==NULL: remove all dynamically allocated (deserialized) data except class instances
soap_delete(**struct** soap *soap, **void** *p)	Remove class instance at p. When p==NULL: remove all dynamically allocated (deserialized) class instances (this is identical to calling soap_destroy(**struct** soap *soap))
soap_unlink(**struct** soap *soap, **void** *p)	Unlink data/object at p from gSOAP's deallocation chain so gSOAP won't deallocate it
soap_done(**struct** soap *soap)	Detach context (reset runtime context)
soap_free(**struct** soap *soap)	Detach and free context (allocated with soap_new)

Temporary data (i.e. the hash tables) are automatically removed with calls to the soap_free_temp function which is also made by soap_end and soap_done or when the next call to a stub or skeleton routine is made to send a message or receive a message. Deallocation of non-class based data is straightforward: soap_end removes all dynamically allocated deserialized data (data allocated with soap_malloc. That is, when the client/service application does not use any class instances that are (de)marshalled, but uses structs, arrays, etc., then calling the soap_end function is safe to remove

all deserialized data. The function can be called after processing the deserialized data of a service operation call or after a number of service operation calls have been made. The function is also typically called after soap_serve, when the service finished sending the response to a client and the deserialized client request data can be removed.

Individual data objects can be unlinked from the deallocation chain if necessary, to prevent deallocation by the collective soap_end or soap_destroy functions.

9.13.1   Memory Allocation and Management Policies

There are three situations to consider for memory deallocation policies for class instances:

1. the program code deletes the class instances and the class destructors in turn SHOULD delete and free any dynamically allocated data (deep deallocation) without calling the soap_end and soap_destroy functions,

2. or the class destructors SHOULD NOT deallocate any data and the soap_end and soap_destroy functions can be called to remove the data.

3. or the class destructors SHOULD mark their own deallocation and mark the deallocation of any other data deallocated by it's destructors by calling the soap_unlink function. This allows soap_destroy and soap_end to remove the remaining instances and data without causing duplicate deallocations.

It is advised to use pointers to class instances that are used within other structs and classes to avoid the creation of temporary class instances during deserialization. The problem with temporary class instances is that the destructor of the temporary may affect data used by other instances through the sharing of data parts accessed with pointers. Temporaries and even whole copies of class instances can be created when deserializing SOAP multi-referenced objects. A dynamic array of class instances is similar: temporaries may be created to fill the array upon deserialization. To avoid problems, use dynamic arrays of pointers to class instances. This also enables the exchange of polymorphic arrays when the elements are instances of classes in an inheritance hierarchy. In addition, allocate data and class instances with soap_malloc and soap_new_X functions (more details below).

To summarize, it is advised to pass class data types by pointer to a service operation. For example:

```
class X { ... };
ns__remoteMethod(X *in, ...);
```

Response elements that are class data types can be passed by reference, as in:

```
class X { ... };
class ns__remoteMethodResponse { ... };
ns__remoteMethod(X *in, ns__remoteMethodResponse &out);
```

But dynamic arrays declared as class data types should use a pointer to a valid object that will be overwritten when the function is called, as in:

```
typedef int xsd__int;
class X { ... };
class ArrayOfint { xsd__int *__ptr; int __size; };
ns__remoteMethod(X *in, ArrayOfint *out);
```

Or a reference to a valid or NULL pointer, as in:

```
typedef int xsd__int;
class X { ... };
class ArrayOfint { xsd__int *__ptr; int __size; };
ns__remoteMethod(X *in, ArrayOfint *&out);
```

The gSOAP memory allocation functions can be used in client and/or service code to allocate temporary data that will be automatically deallocated. These functions are:

Function Call	Description
void *soap_malloc(**struct** soap *soap, size_t n)	return pointer to n bytes
<u>Class</u> *soap_new_<u>Class</u>(**struct** soap *soap)	instantiate <u>Class</u>
<u>Class</u> *soap_new_<u>Class</u>(**struct** soap *soap, **int** n)	instantiate array of n objects
<u>Class</u> *soap_new_set_<u>Class</u>(**struct** soap *soap, m_1, ..., m_n)	instantiate and set members m_i
<u>Class</u> *soap_new_req_<u>Class</u>(**struct** soap *soap, m_1, ..., m_n)	instantiate and set required-only m_i

The soap_new_X functions are generated by the gSOAP soapcpp2 compiler for every class X in the header file.

Space allocated with soap_malloc will be released with the soap_end and soap_dealloc functions. All objects instantiated with soap_new_X(**struct** soap*) are removed altogether with soap_destroy(**struct** soap*). To remove just a single object, use soap_delete_X(**struct** soap*, X*).

For example, the following service uses temporary data in the service operation implementation:

```
int main()
{ ...
    struct soap soap;
    soap_init(&soap);
    soap_serve(&soap);
    soap_end(&soap);
    ...
}
```

An example service operation that allocates a temporary string is:

```
int ns__itoa(struct soap *soap, int i, char **a)
{
    *a = (char*)soap_malloc(soap, 11);
    sprintf(*a, "%d", i);
    return SOAP_OK;
}
```

This temporary allocation can also be used to allocate strings for the SOAP Fault data structure. For example:

```
int ns__mymethod(...)
{ ...
  if (exception)
  {
    char *msg = (char*)soap_malloc(soap, 1024); // allocate temporary space for detailed message
    sprintf(msg, "...", ...); // produce the detailed message
    return soap_receiver_fault(soap, "An exception occurred", msg); // return the server-side fault
  }
  ...
}
```

Use soap_receiver_fault(**struct** soap *soap, **const char** *faultstring, **const char** *detail) to set a SOAP 1.1/1.2 fault at the server-side. Use soap_sender_fault(**struct** soap *soap, **const char** *faultstring, **const char** *detail) to set a SOAP 1.1/1.2 unrecoverable Bad Request fault at the server-side. Sending clients are not supposed to retry messages after a Bad Request, while errors at the receiver-side indicate temporary problems.

The above functions do not include a SOAP 1.2 Subcode element. To include Subcode element, use soap_receiver_fault_subcode(**struct** soap *soap, **const char** *subcode, **const char** *faultstring, **const char** *detail) to set a SOAP 1.1/1.2 fault with Subcode at the server-side. Use soap_sender_fault_subcode(**struct** soap *soap, **const char** *subcode, **const char** *faultstring, **const char** *detail) to set a SOAP 1.1/1.2 unrecoverable Bad Request fault with Subcode at the server-side.

gSOAP provides a function to duplicate a string into gSOAP's memory space:

> **char** *soap_strdup(**struct** soap *soap, const char *s)

The function allocates space for s with soap_malloc, copies the string, and returns a pointer to the duplicated string. When s is NULL, the function does not allocate and copy the string and returns NULL.

9.13.2   Intra-Class Memory Management

When a class declaration has a **struct** soap * field, this field will be set to point to the current gSOAP runtime context by gSOAP's deserializers and by the soap_new_Class functions. This simplifies memory management for class instances. The **struct** soap* pointer is implicitly set by the gSOAP deserializer for the class or explicitly by calling the soap_new_X function for class X. For example:

```
class Sample
{ public:
  struct soap *soap; // reference to gSOAP's run-time
  ...
  Sample();
  ~Sample();
};
```

The constructor and destructor for class Sample are:

```
Sample::Sample()
{ this->soap = NULL;
}
Sample::~Sample()
{ soap_unlink(this->soap, this);
}
```

The soap_unlink() call removes the object from gSOAP's deallocation chain. In that way, soap_destroy
can be safely called to remove all class instances. The following code illustrates the explicit creation
of a Sample object and cleanup:

```
struct soap *soap = soap_new(); // new gSOAP runtime
Sample *obj = soap_new_Sample(soap); // new Sample object with obj->soap set to runtime
...
delete obj; // also calls soap_unlink to remove obj from the deallocation chain
soap_destroy(soap); // deallocate all (other) class instances
soap_end(soap); // clean up
```

Here is another example:

```
class ns__myClass
{ ...
    struct soap *soap; // set by soap_new_ns__myClass()
    char *name;
    void setName(const char *s);
    ...
};
```

Calls to soap_new_ns__myClass(soap) will set the soap field in the class instance to the current gSOAP
context. Because the deserializers invoke the soap_new functions, the soap field of the ns__myClass
instances are set as well. This mechanism is convenient when Web Service methods need to return
objects that are instantiated in the methods. For example

```
int ns__myMethod(struct soap *soap, ...)
{
    ns__myClass *p = soap_new_ns__myClass(soap);
    p->setName("SOAP");
    return SOAP_OK;
}
void ns__myClass::ns__setName(const char *s)
{
    if (soap)
        name = (char*)soap_malloc(soap, strlen(s)+1);
    else
        name = (char*)malloc(strlen(s)+1);
    strcpy(name, s);
}
ns__myClass::ns__myClass()
{
    soap = NULL;
```

```
    name = NULL;
}
ns_ _myClass::~ns_ _myClass()
{
    if (!soap && name) free(name);
    soap_unlink(soap, this);
}
```

Calling soap_destroy right after soap_serve in the Web Service will destroy all dynamically allocated class instances.

9.14 Debugging

To activate debugging and message logging, set the #define DEBUG macro on the compiler's command line (typically as a compiler option -DDEBUG) or in stdsoap2.h, and recompile your code together with stdsoap2.c or stdsoap2.cpp (instead of libgsoap). When using libgsoap and libgsoap++, reinstall the software with configure using option –enable-debug.

When your client and server applications run, they will log their activity in three separate files:

File	Description
SENT.log	The SOAP content transmitted by the application
RECV.log	The SOAP content received by the application
TEST.log	A log containing various activities performed by the application

Caution: The client and server applications may run slow due to the logging activity.

Hint: Set macro DEBUG_STAMP instead of DEBUG to add time stamps to TEST.log. This works on platforms supporting the gettimeofday function.

Caution: When installing a CGI application on the Web with debugging activated, the log files may sometimes not be created due to file access permission restrictions imposed on CGI applications. To get around this, create empty log files with universal write permissions. Be careful about the security implication of this.

You can test a service CGI application without deploying it on the Web. To do this, create a client application for the service and activate message logging by this client. Remove any old SENT.log file and run the client (which connects to the Web service or to another dummy, but valid address) and copy the SENT.log file to another file, e.g. SENT.tst. Then redirect the SENT.tst file to the service CGI application. For example,

> ./myservice.cgi < SENT.tst

This should display the service response on the terminal.

The file names of the log files and the logging activity can be controlled at the application level. This allows the creation of separate log files by separate services, clients, and threads. For example, the following service logs all SOAP messages (but no debug messages) in separate directories:

```
    struct soap soap;
    soap_init(&soap);
```

```
...
soap_set_recv_logfile(&soap, "logs/recv/service12.log"); // append all messages received in /logs/recv/service12.log
soap_set_sent_logfile(&soap, "logs/sent/service12.log"); // append all messages sent in /logs/sent/service12.log
soap_set_test_logfile(&soap, NULL); // no file name: do not save debug messages
...
soap_serve(&soap);
...
```

Likewise, messages can be logged for individual client-side service operation calls.

9.15   Generating an Auto Test Server for Client Testing

The soapcpp2 -T option generates an auto-test server application in soapTester.cpp, which is to be compiled and linked with the code generated for a server implementation, i.e. soapServer.cpp (or with the generated server object class) and soapC.cpp. The feature also supports C, so use the soapcpp2 -c option to generate C.

The auto-test server can be used to test a client application. Suppose the generated code is compiled into the executable named tester (compile soapServer.cpp, soapC.cpp, and stdsoap2.cpp or link libgsoap++). We can use the IO redirect to "send" it a message saved in a file, for example one of the sample request messages generated by soapcpp2:

> ./tester < example.req.xml

which then returns the response with default XML values displayed on the terminal.

To run the auto test service on a port to test a client against, use two command-line arguments. The first argument is the OR-ed values of the gSOAP runtime context flags such as SOAP_IO_KEEPALIVE (0x10 = 16) and the second argument is the port number:

> ./tester 16 8080

This starts an iterative stand-alone server on port 8080. This way, messages can be sent to http://localhost:8080 to test the client. The data in the response messages are copied from the request messages when possible, or XML default values, or empty otherwise.

9.16   Required Libraries

- The socket library is essential and requires the inclusion of the appropriate libraries with the compile command for Sun Solaris systems:

    > c++ -o myclient myclient.cpp stdsoap2.cpp soapC.cpp soapClient.cpp -lsocket -lxnet -lnsl

  These library loading options are not required with Linux.

- The gSOAP runtime uses the math library for the NaN, INF, and -INF floating point representations. The library is not strictly necessary and the <math.h> header file import can be commented out from the stdsoap2.h header file. The application can be linked without the -lm math library e.g. under Sun Solaris:

    > c++ -o myclient myclient.cpp stdsoap2.cpp soapC.cpp soapClient.cpp -lsocket -lxnet -lnsl

10   The gSOAP Service Operation Specification Format

A service operation is specified as a C/C++ function prototype in a header file. The function is REQUIRED to return **int**, which is used to represent a SOAP error code, see Section 10.2. Multiple service operations MAY be declared together in one header file.

The general format of a service operation specification is:

[**int**] [namespace_prefix_ _]method_name([inparam1, inparam2, ...,] outparam);

where

namespace_prefix_ _ is the optional namespace prefix of the method (see identifier translation rules 10.3)

method_name it the service operation name (see identifier translation rules 10.3)

inparam is the declaration of an input parameter of the service operation

outparam is the declaration of the output parameter of the service operation

This simple form can only pass a single, non-**struct** and non-**class** type output parameter. See 10.1 for passing multiple output parameters. The name of the declared function namespace_prefix_ _method_name must be unique and cannot match the name of a **struct**, **class**, or **enum** declared in the same header file.

The method request is encoded in SOAP as an XML element and the namespace prefix, method name, and input parameters are encoded using the format:

```
<[namespace-prefix:]method_name xsi:type="[namespace-prefix:]method_name>
<inparam-name1 xsi:type="...">...</inparam-name1>
<inparam-name2 xsi:type="...">...</inparam-name2>
...
</[namespace-prefix:]method_name>
```

where the `inparam-name` accessors are the element-name representations of the inparam parameter name declarations, see Section 10.3. (The optional parts are shown enclosed in [].)

The XML response by the Web service is of the form:

```
<[namespace-prefix:]method-nameResponse xsi:type="[namespace-prefix:]method-nameResponse>
<outparam-name xsi:type="...">...</outparam-name>
</[namespace-prefix:]method-nameResponse>
```

where the `outparam-name` accessor is the element-name representation of the outparam parameter name declaration, see Section 10.3. By convention, the response element name is the method name ending in `Response`. See 10.1 on how to change the declaration if the service response element name is different.

The gSOAP soapcpp2 tool generates a stub routine for the service operation. This stub is of the form:

int soap_call_[namespace_prefix__]method_name(**struct** soap *soap, **char** *URL, **char** *action, [inparam1, inparam2, ...,] outparam);

This proxy can be called by a client application to perform the service operation call.

The gSOAP `soapcpp2` tool generates a skeleton routine for the service operation. The skeleton function is:

int soap_serve_[namespace_prefix__]method_name(**struct** soap *soap);

The skeleton routine, when called by a service application, will attempt to serve a request on the standard input. If no request is present or if the request does not match the method name, SOAP_NO_METHOD is returned. The skeleton routines are automatically called by the generated soap_serve routine that handles all requests.

10.1 Service Operation Parameter Passing

The input parameters of a service operation MUST be passed by value. Input parameters cannot be passed by reference with the & reference operator, but an input parameter value MAY be passed by a pointer to the data. Of course, passing a pointer to the data is preferred when the size of the data of the parameter is large. Also, to pass instances of (derived) classes, pointers to the instance need to be used to avoid passing the instance by value which requires a temporary and prohibits passing derived class instances. When two input parameter values are identical, passing them using a pointer has the advantage that the value will be encoded only once as multi-reference (hence, the parameters are aliases). When input parameters are passed using a pointer, the data pointed to will not be modified by the service operation and returned to the caller.

The output parameter MUST be passed by reference using & or by using a pointer. Arrays are passed by reference by default and do not require the use of the reference operator &.

The input and output parameter types have certain limitations, see Section 9.10

If the output parameter is a **struct** or **class** type, it is considered a service operation response element instead of a simple output parameter value. That is, the name of the **struct** or **class** is the name of the response element and the **struct** or **class** fields are the output parameters of the service operation, see also 7.1.7. Hence, if the output parameter has to be a **struct** or **class**, a response **struct** or **class** MUST be declared as well. In addition, if a service operation returns multiple output parameters, a response **struct** or **class** MUST be declared. By convention, the response element is the service operation name ending with "`Response`".

The general form of a response element declaration is:

```
struct [namespace_prefix__]response_element_name
{
  outparam1;
  outparam2;
  ...
};
```

where

namespace_prefix__ is the optional namespace prefix of the response element (see identifier translation rules 10.3)

response_element_name it the name of the response element (see identifier translation rules 10.3)

outparam is the declaration of an output parameter of the service operation

The general form of a service operation specification with a response element declaration for (multiple) output parameters is:

> [**int**] [namespace_prefix__]method_name([inparam1, inparam2, ...,] **struct** [namespace_prefix__]response_element_name {outparam1[, outparam2, ...]} &anyparam);

The choice of name for anyparam has no effect on the SOAP encoding and decoding and is only used as a place holder for the response.

The method request is encoded in SOAP as an independent element and the namespace prefix, method name, and input parameters are encoded using the format:

```
<[namespace-prefix:]method-name xsi:type="[namespace-prefix:]method-name>
<inparam-name1 xsi:type="...">...</inparam-name1>
<inparam-name2 xsi:type="...">...</inparam-name2>
...
</[namespace-prefix:]method-name>
```

where the `inparam-name` accessors are the element-name representations of the inparam parameter name declarations, see Section 10.3. (The optional parts resulting from the specification are shown enclosed in [].)

The method response is expected to be of the form:

```
<[namespace-prefix:]response-element-name xsi:type="[namespace-prefix:]response-element-name>
<outparam-name1 xsi:type="...">...</outparam-name1>
<outparam-name2 xsi:type="...">...</outparam-name2>
...
</[namespace-prefix:]response-element-name>
```

where the `outparam-name` accessors are the element-name representations of the outparam parameter name declarations, see Section 10.3. (The optional parts resulting from the specification are shown enclosed in [].)

The input and/or output parameters can be made anonymous, which allows the deserialization of requests/responses with different parameter names as is endorsed by the SOAP 1.1 specification, see Section 7.1.13.

10.2   Error Codes

The error codes returned by the stub and skeleton routines are listed below.

Code	Description
SOAP_OK	No error
SOAP_CLI_FAULT*	The service returned a client fault (SOAP 1.2 Sender fault)
SOAP_SVR_FAULT*	The service returned a server fault (SOAP 1.2 Receiver fault)
SOAP_TAG_MISMATCH	An XML element didn't correspond to anything expected
SOAP_TYPE	An XML Schema type mismatch
SOAP_SYNTAX_ERROR	An XML syntax error occurred on the input
SOAP_NO_TAG	Begin of an element expected, but not found
SOAP_IOB	Array index out of bounds
SOAP_MUSTUNDERSTAND*	An element needs to be ignored that need to be understood
SOAP_NAMESPACE	Namespace name mismatch (validation error)
SOAP_FATAL_ERROR	Internal error
SOAP_USER_ERROR	User error (reserved for soap.user usage
SOAP_FAULT	An exception raised by the service
SOAP_NO_METHOD	The dispatcher did not find a matching operation for a request
SOAP_NO_DATA	No data in HTTP message
SOAP_GET_METHOD	HTTP GET operation not handled, see Section 19.10
SOAP_EOM	Out of memory
SOAP_MOE	Memory overflow/corruption error (DEBUG mode)
SOAP_NULL	An element was null, while it is not supposed to be null
SOAP_DUPLICATE_ID	Element's ID duplicated (multi-ref encoding)
SOAP_MISSING_ID	Element ID missing for an href/ref (multi-ref encoding)
SOAP_HREF	Reference to object is incompatible with the object refered to
SOAP_UTF_ERROR	An UTF-encoded message decoding error occured
SOAP_UDP_ERROR	Message too large to store in UDP packet
SOAP_TCP_ERROR	A connection error occured
SOAP_HTTP_ERROR	An HTTP error occured
SOAP_NTLM_ERROR	An NTLM authentication handshake error occured
SOAP_SSL_ERROR	An SSL error occured
SOAP_ZLIB_ERROR	A Zlib error occured
SOAP_PLUGIN_ERROR	Failed to register plugin
SOAP_MIME_ERROR	MIME parsing error
SOAP_MIME_HREF	MIME attachment has no href from SOAP body error
SOAP_MIME_END	End of MIME attachments protocol error
SOAP_DIME_ERROR	DIME formatting error or DIME size exceeds SOAP_MAXDIMESIZE
SOAP_DIME_END	End of DIME attachments protocol error
SOAP_DIME_HREF	DIME attachment has no href from SOAP body
	(and no DIME callbacks were defined to save the attachment)
SOAP_DIME_MISMATCH	DIME version/transmission error
SOAP_VERSIONMISMATCH*	SOAP version mismatch or no SOAP message
SOAP_DATAENCODINGUNKNOWN	SOAP 1.2 DataEncodingUnknown fault
SOAP_REQUIRED	Attributed required validation error
SOAP_PROHIBITED	Attributed prohibited validation error
SOAP_OCCURS	Element minOccurs/maxOccurs validation error
SOAP_LENGTH	Element length validation error
SOAP_FD_EXCEEDED	Too many open sockets
	(for non-win32 systems not supporting poll())
SOAP_EOF	Unexpected end of file, no input, or timeout receiving data
SOAP_ERR	Error (for internal use)

The error codes that are returned by a stub routine (proxy) upon receiving a SOAP Fault from the server are marked (*). The remaining error codes are generated by the proxy itself as a result of problems with a SOAP payload. The error code is SOAP_OK when the service operation call was successful (the SOAP_OK predefined constant is guaranteed to be 0). The error code is also stored in soap.error, where soap is a variable that contains the current runtime context. The function soap_print_fault(**struct** soap *soap, FILE *fd) can be called to display an error message on fd where current value of the soap.error variable is used by the function to display the error. The function soap_print_fault_location(**struct** soap *soap, FILE *fd) prints the location of the error if the error is a result from parsing XML. Use soap_sprint_fault(**struct** soap*, **char** *buf, size_t len) to print the error to a string.

A service operation implemented in a SOAP service MUST return an error code as the function's return value. SOAP_OK denotes success and SOAP_FAULT denotes an exception. The exception details can be assigned with the soap_receiver_fault(**struct** soap *soap, **const char** *faultstring, **const char** *detail) which sets the strings soap.fault−>faultstring and soap.fault−>detail for SOAP 1.1, and soap.fault−>SOAP_ENV__Reason and soap.fault−>SOAP_ENV__Detail for SOAP 1.2, where soap is a variable that contains the current runtime context, see Section 12. A receiver error indicates that the service can't handle the request, but can possibly recover from the error. To return an unrecoverable SOAP 1.1/1.2 error, use soap_sender_fault(**struct** soap *soap, **const char** *faultstring, **const char** *detail).

To return a HTTP error code a service method can simply return the HTTP error code number. For example, **return** 404; returns a "404 Not Found" HTTP error back to the client. The soap.error is set to the HTTP error code at the client side. The HTTP 1.1 error codes are:

#	Error
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
307	Temporary Redirect
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Time-out
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Request Entity Too Large
414	Request-URI Too Large
415	Unsupported Media Type
416	Requested range not satisfiable
417	Expectation Failed
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Time-out
505	HTTP Version not supported

The error codes are given for informational purposes only. The HTTP protocol requires the proper actions after an error is issued. gSOAP's HTTP 1.0/1.1 handling is automatic.

10.3 C/C++ Identifier Name to XML Tag Name Mapping

One of the "secrets" behind the power and flexibility of gSOAP's encoding and decoding of service operation names, class names, type identifiers, and struct or class fields is the ability to specify namespace prefixes with these names that are used to denote their encoding style. More specifically, a C/C++ identifier name of the form

[namespace_prefix__]element_name

where the prefix and the element name are separated by double underscores will be encoded in XML as

```
<[namespace-prefix:]element-name ...>
```

The **underscore pair** (_ _) separates the namespace prefix from the element name. Each namespace prefix has a namespace URI specified by a namespace mapping table 10.4, see also Section 7.1.2. The namespace URI is a unique identification that can be associated with the service operations and data types. The namespace URI disambiguates potentially identical service operation names and data type names used by disparate organizations.

XML element names are NCNames (restricted strings) that MAY contain **hyphens**, **dots**, and **underscores**. The special characters in the XML element names of service operations, structs, classes, typedefs, and fields can be controlled using the following conventions: A **single underscore** in a namespace prefix or identifier name is replaced by a hyphen (-) in the XML element name. For example, the identifier name SOAP_ENC_ _ur_type is represented in XML as `SOAP-ENC:ur-type`. The sequence _DOT is replaced by a dot (.), and the sequence _USCORE is replaced by an underscore (_) in the corresponding XML element name. For example:

```
class n_s_ _biz_DOTcom
{
   char *n_s_ _biz_USCOREname;
};
```

is encoded in XML as:

```
<n-s:biz.com xsi:type="n-s:biz.com">
   <n-s:biz_name xsi:type="string">Bizybiz</n-s:biz_name>
</n-s:biz.com>
```

Trailing underscores of an identifier name are not translated into the XML representation. This is useful when an identifier name clashes with a C++ keyword. For example, `return` is often used as an accessor name in a SOAP response element. The `return` element can be specified as return_ in the C++ source code. Note that XML should be treated as case sensitive, so the use of e.g. Return may not always work to avoid a name clash with the **return** keyword. The use of trailing underscores also allows for defining **struct**s and **class**es with essentially the same XML Schema type name, but that have to be distinguished as seperate C/C++ types.

For decoding, the underscores in identifier names act as wildcards. An XML element is parsed and matches the name of an identifier if the name is identical to the element name (case insensitive) and the underscores in the identifier name are allowed to match any character in the element name. For example, the identifier name I_want_ _soap_fun_the_bea_ _ _DOTcom matches the element name `I-want:SOAP4fun@the-beach.com`.

By default, soapcpp2 generates data bindings in which all XML elements are and attributes are unqualified:

```
//gsoap x schema namespace: urn:x
struct x_ _record
```

118

```
{
  @char * type;
  char * name;
};
```

where the name element and the type attribute are unqualified in the XML content (for example to facilitate SOAP RPC encoding).

The rules for SOAP services that are document style are different:

```
//gsoap x schema namespace: urn:x
//gsoap x service style: document
struct x__record
{
  @char * type;
  char * name;
};
```

where x is associated with a service. For document style all elements are qualified and attributes are unqualified.

To force qualification of elements and attributes, use the "form" directive:

```
//gsoap x schema namespace: urn:x
//gsoap x schema form: qualified
struct x__record
{
  @char * type;
  char * name;
};
```

You can also use "elementForm" and "attributeForm" directives to (un)qualify element and attributes of the schema, respectively.

Because the soapcpp2-generated serializers follow the qualified/unqualified forms of the schemas, there is normally no need to explicitly qualify struct/class members because automatic encoding rules will be used.

If explicit qualification is needed, this can be done using the prefix convention:

```
//gsoap x schema namespace: urn:x
//gsoap y schema namespace: urn:y
struct x__record
{
  @char * xsi__type;
  char * y__name;
};
```

which ensures that there cannot be any name clashes between members of the same name defined in different schemas (consider for example name and y__name), but this can clutter the representation when clashes do not occur.

An alternative to the prefix convention is the use of "**colon notation**" in the gSOAP header file. This deviation from the C/C++ syntax allows you to bind type names and struct and class members to qualified and unqualified XML tag names explicitly, thus bypassing the default mechanism that automatically qualifies or unqualifies element and attribute tag names based on the schema element/attribute form.

The colon notation for type names, struct/class names and members overrides the prefix qualification rules explicitly:

```
//gsoap x schema namespace: urn:x
//gsoap y schema namespace: urn:y
struct x:record
{
  @char * xsi:type;
  char * y:name;
};
```

where x and y are namespace prefixes that MUST be declared with a directive. The xsi:type member is an XML attribute in the xsi namespace. The soapcpp2 tool maps this to the following struct without the annotations:

```
// This code is generated from the above by soapcpp2 in soapStub.h:
struct record
{
  char *type; /* optional attribute of type xsd:string */
  char *name; /* optional element of type xsd:string */
};
```

The soapcpp2 tool also generates XML schemas with element and attribute references. That is, y:name is referenced from the y schema by the `x:record` complexType defined in the x schema.

The colon notation also allows you to override the element/attribute form to unqualified for qualified schemas:

```
//gsoap x schema namespace: urn:x
//gsoap x schema form: qualified
struct x:record
{
  @char * :type;
  char * :name;
};
```

where the colon notation ensures that both type and name are unqualified in the XML content, which overrides the default qualified forms of the x schema.

Note that the use of colon notation to bind namespace prefixes to type names (typedef, enum, struct, and class names) translates to code without the prefixes. This means that name clashes can occur between types with identical unquaified names:

```
enum x:color { RED, WHITE, BLUE };
enum y:color { YELLOW, ORANGE }; // illegal enum name: name clash with x:color
```

120

while prefixing with double underscores never lead to clashes:

```
enum x_ _color { RED, WHITE, BLUE };
enum y_ _color { YELLOW, ORANGE }; // no name clash
```

Also note that colon notation has a very different role than the C++ scope operator ::. The scope operator cannot be used in places where we need colon notation, such as struct/class member fields.

The default mechanism that associates XML tag names with the names of struct and class member fields can be overriden by "**retagging**" names with the annotation of '*tag*' placed next to the member field name. This is particularly useful to support legacy code for which the fixed naming of member fields cannot be easily changed. For example:

```
//gsoap x schema namespace: urn:x
//gsoap x schema form: qualified
struct x:record
{
  @char * t'type';
  char * s'full-name';
};
```

This maps the `t` member to the `x:type` XML attribute tag and `s` member to the `x:full-name` XML element tag. Note that both tags are namespace qualified as per schema declaration.

10.4   Namespace Mapping Table

A namespace mapping table MUST be defined by clients and service applications. The mapping table is used by the serializers and deserializers of the stub and skeleton routines to produce a valid SOAP payload and to validate an incoming SOAP payload. A typical mapping table is shown below:

```
struct Namespace namespaces[] =
{   // {"ns-prefix", "ns-name"}
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"}, // MUST be first
   {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"}, // MUST be second
   {"xsi", "http://www.w3.org/2001/XMLSchema-instance"}, // MUST be third
   {"xsd", "http://www.w3.org/2001/XMLSchema"}, // Required for XML Schema types
   {"ns1", "urn:my-service-URI"}, // The namespace URI of the service operations
   {NULL, NULL} // end of table
};
```

Each namespace prefix used by a identifier name in the header file specification (see Section 10.3) MUST have a binding to a namespace URI in the mapping table. The end of the namespace mapping table MUST be indicated by the NULL pair. The namespace URI matching is case insensitive. A namespace prefix is distinguished by the occurrence of a pair of underscores (_ _) in an identifier.

An optional namespace pattern MAY be provided with each namespace mapping table entry. The patterns provide an alternative namespace matching for the validation of decoded SOAP messages.

In this pattern, dashes (-) are single-character wildcards and asterisks (*) are multi-character wild-cards. For example, to decode different versions of XML Schema type with different authoring dates, four dashes can be used in place of the specific dates in the namespace mapping table pattern:

```
struct Namespace namespaces[] =
{   // {"ns-prefix", "ns-name", "ns-name validation pattern"}
...
   {"xsi", "http://www.w3.org/2001/XMLSchema-instance", "http://www.w3.org/----/XMLSchema-instance"},
   {"xsd", "http://www.w3.org/2001/XMLSchema", "http://www.w3.org/----/XMLSchema"},
...
```

Or alternatively, asterisks can be used as wildcards for multiple characters:

```
struct Namespace namespaces[] =
{   // {"ns-prefix", "ns-name", "ns-name validation pattern"}
...
   {"xsi", "http://www.w3.org/2001/XMLSchema-instance", "http://www.w3.org/*/XMLSchema-instance"},
   {"xsd", "http://www.w3.org/2001/XMLSchema", "http://www.w3.org/*/XMLSchema"},
...
```

A namespace mapping table is automatically generated together with a WSDL file for each namespace prefix that is used for a service operation specified in the header file. This namespace mapping table has entries for all namespace prefixes. The namespace URIs need to be filled in. These appear as http://tempuri.org in the table. See Section 19.2 on how to specify the namespace URIs in the header file.

For decoding elements with namespace prefixes, the namespace URI associated with the namespace prefix (through the `xmlns` attribute of an XML element) is searched from the beginning to the end in a namespace mapping table, and for every row the following tests are performed as part of the validation process:

1. the string in the second column matches the namespace URI (case insensitive)

2. the string in the optional third column matches the namespace URI (case insensitive), where - is a one-character wildcard and * is a multi-character wildcard

When a match is found, the namespace prefix in the first column of the table is considered semantically identical to the namespace prefix used by the XML element to be decoded, though the prefix names may differ. A service will respond with the namespace that it received from a client in case it matches a pattern in the third column.

For example, let's say we have the following structs:

```
struct a__elt { ... };
struct b__elt { ... };
struct k__elt { ... };
```

and a namespace mapping table in the program:

```
struct Namespace namespaces[] =
{   // {"ns-prefix", "ns-name", "ns-name validation pattern"}
...
   {"a", "some uri"},
   {"b", "other uri"},
   {"c", "his uri", "* uri"},
...
```

Then, the following XML elements will match the structs:

```
<n:elt xmlns:n="some URI">      matches the struct name a__elt
...
<m:elt xmlns:m="other URI">      matches the struct name b__elt
...
<k:elt xmlns:k="my URI">     matches the struct name c__elt
...
```

The response of a service to a client request that uses the namespaces listed above, will include `my URI` for the name space of element `k`.

It is possible to use a number of different namespace tables and select the one that is appropriate. For example, an application might contact many different Web services all using different namespace URIs. If all the URIs are stored in one table, each service operation invocation will dump the whole namespace table in the SOAP payload. There is no technical problem with that, but it can be ugly when the table is large. To use different namespace tables, declare a pointer to a table and set the pointer to a particular table before service operation invocation. For example:

```
struct Namespace namespacesTable1[] = { ... };
struct Namespace namespacesTable2[] = { ... };
struct Namespace namespacesTable3[] = { ... };
struct Namespace *namespaces;
...
struct soap soap;
...
soap_init(&soap);
soap_set_namespaces(&soap, namespaceTable1);
soap_call_remote_method(&soap, URL, Action, ...);
...
```

11   gSOAP Serialization and Deserialization Rules

This section describes the serialization and deserialization of C and C++ data types for SOAP 1.1 and 1.2 compliant encoding and decoding.

11.1   SOAP RPC Encoding Versus Document/Literal and xsi:type Info

The wsdl2h tool automatically generates a header file specialized for SOAP RPC encoding or document/literal style. The serialization and deserialization rules for C/C++ objects is almost identical for these styles, except for the following important issues.

With SOAP RPC encoding style, care must be taken to ensure typed messages are produced for interoperability and compatibility reasons. To ensure that the gSOAP engine automatically generates typed (`xsi:type` attributed) messages, use `soapcpp2` option -t, see also Section 9.1. While gSOAP can handle untyped messages, some toolkits fail to find deserializers when the `xsi:type` information is absent.

When starting the development of a gSOAP application from a header file, the `soapcpp2` compiler will generate WSDL and schema files for SOAP 1.1 document/literal style by default (use the //`gsoap` directives to control this, see Section 19.2). Use `soapcpp2` options -2, -e, and -t to generate code for SOAP 1.2, RPC encoding, and typed messages.

With SOAP RPC encoding, generic `complexTypes` with `maxOccurs="unbounded"` are not allowed and SOAP encoded arrays must be used. Also XML attributes and unions (XML schema `choice`) are not allowed with SOAP RPC encoding.

Also with SOAP RPC encoding, multi-reference accessors are common to encode co-referenced objects and object digraphs. Multi-reference encoding is not supported in document/literal style, which means that cyclic object digraphs cannot be serialized (the engine will crash). Also DAGs are represented as XML trees in document/literal style messaging.

11.2   Primitive Type Encoding

The default encoding rules for the primitive C and C++ data types are given in the table below:

Type	XSD Type
bool	`boolean`
char* (C string)	`string`
char	`byte`
long double	`decimal` (with #import "custom/long_double.h")
double	`double`
float	`float`
int	`int`
long	`long`
LONG64	`long`
long long	`long`
short	`short`
time_t	`dateTime`
struct tm	`dateTime` (with #import "custom/struct_tm.h")
unsigned char	`unsignedByte`
unsigned int	`unsignedInt`
unsigned long	`unsignedLong`
ULONG64	`unsignedLong`
unsigned long long	`unsignedLong`
unsigned short	`unsignedShort`
wchar_t*	`string`

Objects of type **void** and **void*** cannot be encoded. Enumerations and bit masks are supported as well, see 11.4.

11.3   How to Represent Primitive C/C++ Types as XSD Types

By default, encoding of the primitive types will take place as per SOAP encoding style. The encoding can be changed to any XML Schema type (XSD type) with an optional namespace prefix by using a **typedef** in the header file input to the gSOAP soapcpp2 tool. The declaration enables the implementation of built-in XML Schema types (also known as XSD types) such as positiveInteger, xsd:anyURI, and xsd:date for which no built-in data structures in C and C++ exist but which can be represented using standard data structures such as strings, integers, and floats.

The **typedef** declaration is frequently used for convenience in C. A **typedef** declares a type name for a (complex) type expression. The type name can then be used in other declarations in place of the more complex type expression, which often improves the readability of the program code.

The gSOAP soapcpp2 compiler interprets **typedef** declarations the same way as a regular C compiler interprets them, i.e. as types in declarations. In addition however, the gSOAP soapcpp2 compiler will also use the type name in the encoding of the data in SOAP. The **typedef** name will appear as the XML element name of an independent element and as the value of the xsi:type attribute in the SOAP payload.

Many built-in primitive and derived XSD types such as xsd:anyURI, positiveInteger, and decimal can be stored by standard primitive data structures in C++, such as strings, integers, floats, and doubles. To serialize strings, integers, floats, and doubles as built-in primitive and derived XSD types, a **typedef** declaration can be used to declare an XSD type.

For example, the declaration

> **typedef unsigned int** xsd_ _positiveInteger;

creates a named type positiveInteger which is represented by **unsigned int** in C++. For example, the encoding of a positiveInteger value 3 is

> `<positiveInteger xsi:type="xsd:positiveInteger">3</positiveInteger>`

The built-in primitive and derived numerical XML Schema types are listed below together with their recommended **typedef** declarations. Note that the SOAP encoding schemas for primitive types are derived from the built-in XML Schema types, so SOAP_ENC_ _ can be used as a namespace prefix instead of xsd_ _.

xsd:anyURI Represents a Uniform Resource Identifier Reference (URI). Each URI scheme imposes specialized syntax rules for URIs in that scheme, including restrictions on the syntax of allowed fragment identifiers. It is recommended to use strings to store xsd:anyURI XML Schema types. The recommended type declaration is:

> **typedef char ***xsd_ _anyURI;

xsd:base64Binary Represents Base64-encoded arbitrary binary data. For using the xsd:base64Binary XSD Schema type, the use of the base64Binary representation of a dynamic array is **strongly**

recommended, see Section 11.12. However, the type can also be declared as a string and the encoding will be string-based:

      typedef char *xsd_ _base64Binary;

With this approach, it is the responsibility of the application to make sure the string content is according to the Base64 Content-Transfer-Encoding defined in Section 6.8 of RFC 2045.

`xsd:boolean` For declaring an `xsd:boolean` XSD Schema type, the use of a bool is **strongly** recommended. If a pure C compiler is used that does not support the bool type, see Section 11.4.5. The corresponding type declaration is:

      typedef bool xsd_ _boolean;

Type xsd_ _boolean declares a Boolean (0 or 1), which is encoded as

      `<xsd:boolean xsi:type="xsd:boolean">...</xsd:boolean>`

`xsd:byte` Represents a byte (-128...127). The corresponding type declaration is:

      typedef char xsd_ _byte;

Type xsd_ _byte declares a byte which is encoded as

      `<xsd:byte xsi:type="xsd:byte">...</xsd:byte>`

`xsd:dateTime` Represents a date and time. The lexical representation is according to the ISO 8601 extended format CCYY-MM-DDThh:mm:ss where "CC" represents the century, "YY" the year, "MM" the month and "DD" the day, preceded by an optional leading "-" sign to indicate a negative number. If the sign is omitted, "+" is assumed. The letter "T" is the date/time separator and "hh", "mm", "ss" represent hour, minute and second respectively. It is recommended to use the time_t type to store `xsd:dateTime` XSD Schema types and the type declaration is:

      typedef time_t xsd_ _dateTime;

However, note that calendar times before the year 1902 or after the year 2037 cannot be represented. Upon receiving a date outside this range, the time_t value will be set to -1.

Strings (**char***) can be used to store `xsd:dateTime` XSD Schema types. The type declaration is:

      typedef char *xsd_ _dateTime;

In this case, it is up to the application to read and set the dateTime representation.

`xsd:date` Represents a date. The lexical representation for date is the reduced (right truncated) lexical representation for dateTime: CCYY-MM-DD. It is recommended to use strings (**char***) to store `xsd:date` XSD Schema types. The type declaration is:

      typedef char *xsd_ _date;

`xsd:decimal` Represents arbitrary precision decimal numbers. It is recommended to use the **double** type to store `xsd:decimal` XSD Schema types and the type declaration is:

> **typedef double** xsd_ _decimal;

Type xsd_ _decimal declares a double floating point number which is encoded as

```
<xsd:double xsi:type="xsd:decimal">...</xsd:double>
```

xsd:double Corresponds to the IEEE double-precision 64-bit floating point type. The type declaration is:

> **typedef double** xsd_ _double;

Type xsd_ _double declares a double floating point number which is encoded as

```
<xsd:double xsi:type="xsd:double">...</xsd:double>
```

xsd:duration Represents a duration of time. The lexical representation for duration is the ISO 8601 extended format PnYn MnDTnH nMnS, where nY represents the number of years, nM the number of months, nD the number of days, T is the date/time separator, nH the number of hours, nM the number of minutes and nS the number of seconds. The number of seconds can include decimal digits to arbitrary precision. It is recommended to use strings (**char***) to store xsd:duration XSD Schema types. The type declaration is:

> **typedef char ***xsd_ _duration;

xsd:float Corresponds to the IEEE single-precision 32-bit floating point type. The type declaration is:

> **typedef float** xsd_ _float;

Type xsd_ _float declares a floating point number which is encoded as

```
<xsd:float xsi:type="xsd:float">...</xsd:float>
```

xsd:hexBinary Represents arbitrary hex-encoded binary data. It has a lexical representation where each binary octet is encoded as a character tuple, consisting of two hexadecimal digits ([0-9a-fA-F]) representing the octet code. For example, "0FB7" is a hex encoding for the 16-bit integer 4023 (whose binary representation is 111110110111. For using the xsd:hexBinary XSD Schema type, the use of the hexBinary representation of a dynamic array is **strongly** recommended, see Section 11.13. However, the type can also be declared as a string and the encoding will be string-based:

> **typedef char ***xsd_ _hexBinary;

With this approach, it is solely the responsibility of the application to make sure the string content consists of a sequence of octets.

xsd:int Corresponds to a 32-bit integer in the range -2147483648 to 2147483647. If the C++ compiler supports 32-bit **int** types, the type declaration can use the **int** type:

> **typedef int** xsd_ _int;

Otherwise, the C++ compiler supports 16-bit **int** types and the type declaration should use the **long** type:

typedef long xsd_‌int;

Type xsd_‌int declares a 32-bit integer which is encoded as

```
<xsd:int xsi:type="xsd:int">...</xsd:int>
```

xsd:integer Corresponds to an unbounded integer. Since C++ does not support unbounded integers as a standard feature, the recommended type declaration is:

typedef long long xsd_‌integer;

Type xsd_‌integer declares a 64-bit integer which is encoded as an unbounded xsd:integer:

```
<xsd:integer xsi:type="xsd:integer">...</xsd:integer>
```

Another possibility is to use strings to represent unbounded integers and do the translation in code.

xsd:long Corresponds to a 64-bit integer in the range -9223372036854775808 to 9223372036854775807. The type declaration is:

typedef long long xsd_‌long;

Or in Visual C++:

typedef LONG64 xsd_‌long;

Type xsd_‌long declares a 64-bit integer which is encoded as

```
<xsd:long xsi:type="xsd:long">...</xsd:long>
```

xsd:negativeInteger Corresponds to a negative unbounded integer (< 0). Since C++ does not support unbounded integers as a standard feature, the recommended type declaration is:

typedef long long xsd_‌negativeInteger;

Type xsd_‌negativeInteger declares a 64-bit integer which is encoded as a xsd:negativeInteger:

```
<xsd:negativeInteger xsi:type="xsd:negativeInteger">...</xsd:negativeInteger>
```

Another possibility is to use strings to represent unbounded integers and do the translation in code.

xsd:nonNegativeInteger Corresponds to a non-negative unbounded integer (> 0). Since C++ does not support unbounded integers as a standard feature, the recommended type declaration is:

typedef unsigned long long xsd_‌nonNegativeInteger;

Type xsd_‌nonNegativeInteger declares a 64-bit unsigned integer which is encoded as a non-negative unbounded xsd:nonNegativeInteger:

```
<xsd:nonNegativeInteger xsi:type="xsd:nonNegativeInteger">...</xsd:nonNegativeInteger>
```

Another possibility is to use strings to represent unbounded integers and do the translation in code.

`xsd:nonPositiveInteger` Corresponds to a non-positive unbounded integer (≤ 0). Since C++ does not support unbounded integers as a standard feature, the recommended type declaration is:

> **typedef long long** xsd_ _nonPositiveInteger;

Type xsd_ _nonPositiveInteger declares a 64-bit integer which is encoded as a `xsd:nonPositiveInteger`:

> `<xsd:nonPositiveInteger xsi:type="xsd:nonPositiveInteger">...</xsd:nonPositiveInteger>`

Another possibility is to use strings to represent unbounded integers and do the translation in code.

`xsd:normalizedString` Represents normalized character strings. Normalized character strings do not contain the carriage return (#xD), line feed (#xA) nor tab (#x9) characters. It is recommended to use strings to store `xsd:normalizeString` XSD Schema types. The type declaration is:

> **typedef char** *xsd_ _normalizedString;

Type xsd_ _normalizedString declares a string type which is encoded as

> `<xsd:normalizedString xsi:type="xsd:normalizedString">...</xsd:normalizedString>`

It is solely the responsibility of the application to make sure the strings do not contain carriage return (#xD), line feed (#xA) and tab (#x9) characters.

`xsd:positiveInteger` Corresponds to a positive unbounded integer (≥ 0). Since C++ does not support unbounded integers as a standard feature, the recommended type declaration is:

> **typedef unsigned long long** xsd_ _positiveInteger;

Type xsd_ _positiveInteger declares a 64-bit unsigned integer which is encoded as a `xsd:positiveInteger`:

> `<xsd:positiveInteger xsi:type="xsd:positiveInteger">...</xsd:positiveInteger>`

Another possibility is to use strings to represent unbounded integers and do the translation in code.

`xsd:short` Corresponds to a 16-bit integer in the range -32768 to 32767. The type declaration is:

> **typedef short** xsd_ _short;

Type xsd_ _short declares a short 16-bit integer which is encoded as

> `<xsd:short xsi:type="xsd:short">...</xsd:short>`

`xsd:string` Represents character strings. The type declaration is:

> **typedef char** *xsd_ _string;

Type xsd_ _string declares a string type which is encoded as

> `<xsd:string xsi:type="xsd:string">...</xsd:string>`

The type declaration for wide character strings is:

> **typedef** wchar_t *xsd_ _string;

Both type of strings can be used at the same time, but requires one typedef name to be changed by appending an underscore which is invisible in XML. For example:

> **typedef** wchar_t *xsd_ _string_;

xsd:time Represents a time. The lexical representation for time is the left truncated lexical representation for dateTime: hh:mm:ss.sss with optional following time zone indicator. It is recommended to use strings (**char***) to store xsd:time XSD Schema types. The type declaration is:

> **typedef char** *xsd_ _time;

xsd:token Represents tokenized strings. Tokens are strings that do not contain the line feed (#xA) nor tab (#x9) characters, that have no leading or trailing spaces (#x20) and that have no internal sequences of two or more spaces. It is recommended to use strings to store xsd:token XSD Schema types. The type declaration is:

> **typedef char** *xsd_ _token;

Type xsd_ _token declares a string type which is encoded as

> `<xsd:token xsi:type="xsd:token">...</xsd:token>`

It is solely the responsibility of the application to make sure the strings do not contain the line feed (#xA) nor tab (#x9) characters, that have no leading or trailing spaces (#x20) and that have no internal sequences of two or more spaces.

xsd:unsignedByte Corresponds to an 8-bit unsigned integer in the range 0 to 255. The type declaration is:

> **typedef unsigned char** xsd_ _unsignedByte;

Type xsd_ _unsignedByte declares a unsigned 8-bit integer which is encoded as

> `<xsd:unsignedByte xsi:type="xsd:unsignedByte">...</xsd:unsignedByte>`

xsd:unsignedInt Corresponds to a 32-bit unsigned integer in the range 0 to 4294967295. If the C++ compiler supports 32-bit **int** types, the type declaration can use the **int** type:

> **typedef unsigned int** xsd_ _unsignedInt;

Otherwise, the C++ compiler supports 16-bit **int** types and the type declaration should use the **long** type:

> **typedef unsigned long** xsd_ _unsignedInt;

Type xsd_ _unsignedInt declares an unsigned 32-bit integer which is encoded as

> `<xsd:unsignedInt xsi:type="xsd:unsignedInt">...</xsd:unsignedInt>`

xsd:unsignedLong Corresponds to a 64-bit unsigned integer in the range 0 to 18446744073709551615. The type declaration is:

> **typedef unsigned long long** xsd__unsignedLong;

Or in Visual C++:

> **typedef** ULONG64 xsd__unsignedLong;

Type xsd__unsignedLong declares an unsigned 64-bit integer which is encoded as

```
<xsd:unsignedLong xsi:type="xsd:unsignedLong">...</xsd:unsignedLong>
```

`xsd:unsignedShort` Corresponds to a 16-bit unsigned integer in the range 0 to 65535. The type declaration is:

> **typedef unsigned short** xsd__unsignedShort;

Type xsd__unsginedShort declares an unsigned short 16-bit integer which is encoded as

```
<xsd:unsignedShort xsi:type="xsd:unsignedShort">...</xsd:unsignedShort>
```

Other XSD Schema types such as `gYearMonth`, `gYear`, `gMonthDay`, `gDay`, `xsd:gMonth`, `QName`, `NOTATION`, etc., can be encoded similarly using a **typedef** declaration.

11.3.1 How to Use Multiple C/C++ Types for a Single Primitive XSD Type

Trailing underscores (see Section 10.3) can be used in the type name in a **typedef** to enable the declaration of multiple storage formats for a single XML Schema type. For example, one part of a C/C++ application's data structure may use plain strings while another part may use wide character strings. To enable this simultaneous use, declare:

> **typedef char** *xsd__string;
> **typedef** wchar_t *xsd__string_;

Now, the xsd__string and xsd__string_ types will both be encoded and decoded as XML string types and the use of trailing underscores allows multiple declarations for a single XML Schema type.

11.3.2 How to use C++ Wrapper Classes to Specify Polymorphic Primitive Types

SOAP 1.1 supports polymorphic types, because XSD Schema types form a hierarchy. The root of the hierarchy is called `xsd:anyType` (`xsd:ur-type` in the older 1999 schema). So, for example, an array of `xsd:anyType` in SOAP may actually contain any mix of element types that are the derived types of the root type. The use of polymorphic types is indicated by the WSDL and schema descriptions of a Web service and can therefore be predicted/expected for each particular case.

On the one hand, the **typedef** construct provides a convenient way to associate C/C++ types with XML Schema types and makes it easy to incorporate these types in a (legacy) C/C++ application. However, on the other hand the **typedef** declarations cannot be used to support polymorphic XML Schema types. Most SOAP clients and services do not use polymorphic types. In case they do, the primitive polymorphic types can be declared as a hierarchy of C++ **class**es that can be used simultaneously with the **typedef** declarations.

The general form of a primitive type declaration that is derived from a super type is:

```
class xsd_ _type_name: [public xsd_ _super_type_name]
{ public: Type _ _item;
   [public:] [private] [protected:]
   method1;
   method2;
   ...
};
```

where <u>Type</u> is a primitive C type. The _ _item field MUST be the first field in this wrapper class.

For example, the XML Schema type hierarchy can be copied to C++ with the following declarations:

```
class xsd_ _anyType { };
class xsd_ _anySimpleType: public xsd_ _anyType { };
typedef char *xsd_ _anyURI;
class xsd_ _anyURI_: public xsd_ _anySimpleType { public: xsd_ _anyURI _ _item; };
typedef bool xsd_ _boolean;
class xsd_ _boolean_: public xsd_ _anySimpleType { public: xsd_ _boolean _ _item; };
typedef char *xsd_ _date;
class xsd_ _date_: public xsd_ _anySimpleType { public: xsd_ _date _ _item; };
typedef time_t xsd_ _dateTime;
class xsd_ _dateTime_: public xsd_ _anySimpleType { public: xsd_ _dateTime _ _item; };
typedef double xsd_ _double;
class xsd_ _double_: public xsd_ _anySimpleType { public: xsd_ _double _ _item; };
typedef char *xsd_ _duration;
class xsd_ _duration_: public xsd_ _anySimpleType { public: xsd_ _duration _ _item; };
typedef float xsd_ _float;
class xsd_ _float_: public xsd_ _anySimpleType { public: xsd_ _float _ _item; };
typedef char *xsd_ _time;
class xsd_ _time_: public xsd_ _anySimpleType { public: xsd_ _time _ _item; };
typedef char *xsd_ _decimal;
class xsd_ _decimal_: public xsd_ _anySimpleType { public: xsd_ _decimal _ _item; };
typedef char *xsd_ _integer;
class xsd_ _integer_: public xsd_ _decimal_ { public: xsd_ _integer _ _item; };
typedef LONG64 xsd_ _long;
class xsd_ _long_: public xsd_ _integer_ { public: xsd_ _long _ _item; };
typedef long xsd_ _int;
class xsd_ _int_: public xsd_ _long_ { public: xsd_ _int _ _item; };
typedef short xsd_ _short;
class xsd_ _short_: public xsd_ _int_ { public: xsd_ _short _ _item; };
typedef char xsd_ _byte;
class xsd_ _byte_: public xsd_ _short_ { public: xsd_ _byte _ _item; };
typedef char *xsd_ _nonPositiveInteger;
class xsd_ _nonPositiveInteger_: public xsd_ _integer_ { public: xsd_ _nonPositiveInteger _ _item; };
typedef char *xsd_ _negativeInteger;
class xsd_ _negativeInteger_: public xsd_ _nonPositiveInteger_ { public: xsd_ _negativeInteger _ _item;
};
typedef char *xsd_ _nonNegativeInteger;
class xsd_ _nonNegativeInteger_: public xsd_ _integer_ { public: xsd_ _nonNegativeInteger _ _item; };
typedef char *xsd_ _positiveInteger;
class xsd_ _positiveInteger_: public xsd_ _nonNegativeInteger_ { public: xsd_ _positiveInteger _ _item;
};
```

```
typedef ULONG64 xsd__unsignedLong;
class xsd__unsignedLong_: public xsd__nonNegativeInteger_ { public: xsd__unsignedLong __item;
};
typedef unsigned long xsd__unsignedInt;
class xsd__unsignedInt_: public xsd__unsginedLong_ { public: xsd__unsignedInt __item; };
typedef unsigned short xsd__unsignedShort;
class xsd__unsignedShort_: public xsd__unsignedInt_ { public: xsd__unsignedShort __item; };
typedef unsigned char xsd__unsignedByte;
class xsd__unsignedByte_: public xsd__unsignedShort_ { public: xsd__unsignedByte __item; };
typedef char *xsd__string;
class xsd__string_: public xsd__anySimpleType { public: xsd__string __item; };
typedef char *xsd__normalizedString;
class xsd__normalizedString_: public xsd__string_ { public: xsd__normalizedString __item; };
typedef char *xsd__token;
class xsd__token_: public xsd__normalizedString_ { public: xsd__token __item; };
```

Note the use of the trailing underscores for the **class** names to distinguish the **typedef** type names from the **class** names. Only the most frequently used built-in schema types are shown. It is also allowed to include the xsd:base64Binary and xsd:hexBinary types in the hierarchy:

```
class xsd__base64Binary: public xsd__anySimpleType { public: unsigned char *__ptr; int __size;
};
class xsd__hexBinary: public xsd__anySimpleType { public: unsigned char *__ptr; int __size; };
```

See Sections 11.12 and 11.13.

Methods are allowed to be added to the classes above, such as constructors and getter/setter methods, see Section 11.6.4.

Wrapper structs are supported as well, similar to wrapper classes. But they cannot be used to implement polymorphism. Rather, the wrapper structs facilitate the use of XML attributes with a primitive typed object, see 11.6.7.

11.3.3 XSD Schema Type Decoding Rules

The decoding rules for the primitive C and C++ data types is given in the table below:

Type	Allows Decoding of	Precision Lost?
bool	[xsd:]boolean	no
char* (C string)	any type, see 11.3.5	no
wchar_t ***** (wide string)	any type, see 11.3.5	no
double	[xsd:]double	no
	[xsd:]float	no
	[xsd:]long	no
	[xsd:]int	no
	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedLong	no
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
	[xsd:]decimal	possibly
	[xsd:]integer	possibly
	[xsd:]positiveInteger	possibly
	[xsd:]negativeInteger	possibly
	[xsd:]nonPositiveInteger	possibly
	[xsd:]nonNegativeInteger	possibly
float	[xsd:]float	no
	[xsd:]long	no
	[xsd:]int	no
	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedLong	no
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
	[xsd:]decimal	possibly
	[xsd:]integer	possibly
	[xsd:]positiveInteger	possibly
	[xsd:]negativeInteger	possibly
	[xsd:]nonPositiveInteger	possibly
	[xsd:]nonNegativeInteger	possibly
long long	[xsd:]long	no
	[xsd:]int	no
	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedLong	possibly
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
	[xsd:]integer	possibly
	[xsd:]positiveInteger	possibly
	[xsd:]negativeInteger	possibly
	[xsd:]nonPositiveInteger	possibly
	[xsd:]nonNegativeInteger	possibly

Type	Allows Decoding of	Precision Lost?
long	[xsd:]long	possibly, if **long** is 32 bit
	[xsd:]int	no
	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedLong	possibly
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
int	[xsd:]int	no
	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedInt	possibly
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
short	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
char	[xsd:]byte	no
	[xsd:]unsignedByte	possibly
unsigned long long	[xsd:]unsignedLong	no
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
	[xsd:]positiveInteger	possibly
	[xsd:]nonNegativeInteger	possibly
unsigned long	[xsd:]unsignedLong	possibly, if **long** is 32 bit
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
unsigned int	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
unsigned short	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
unsigned char	[xsd:]unsignedByte	no
time_t	[xsd:]dateTime	no(?)

Due to limitations in representation of certain primitive C++ types, a possible loss of accuracy may occur with the decoding of certain XSD Schema types as is indicated in the table. The table does not indicate the possible loss of precision of floating point values due to the textual representation of floating point values in SOAP.

All explicitly declared XSD Schema encoded primitive types adhere to the same decoding rules. For example, the following declaration:

> **typedef unsigned long long** xsd__nonNegativeInteger;

enables the encoding and decoding of `xsd:nonNegativeInteger` XSD Schema types (although decoding takes place with a possible loss of precision). The declaration also allows decoding of `xsd:positiveInteger` XSD Schema types, because of the storage as a **unsigned long long** data type.

11.3.4   Multi-Reference Strings

If more than one **char** pointer points to the same string, the string is encoded as a multi-reference value. Consider for example

> **char** *s = "hello", *t = s;

The s and t variables are assigned the same string, and when serialized, t refers to the content of s:

```
<string id="123" xsi:type="string">hello</string>
...
<string href="#123"/>
```

The example assumed that s and t are encoded as independent elements.

Note: the use of **typedef** to declare a string type such as xsd__string will not affect the multi-reference string encoding. However, strings declared with different **typedef**s will never be considered multi-reference even when they point to the same string. For example

> **typedef char** *xsd__string;
> **typedef char** *xsd__anyURI;
> xsd__anyURI *s = "http://www.myservice.com";
> xsd__string *t = s;

The variables s and t point to the same string, but since they are considered different types their content will not be shared in the SOAP payload through a multi-referenced string.

11.3.5   "Smart String" Mixed-Content Decoding

The implementation of string decoding in gSOAP allows for mixed content decoding. If the SOAP payload contains a complex data type in place of a string, the complex data type is decoded in the string as plain XML text.

For example, suppose the getInfo service operation returns some detailed information. The service operation is declared as:

```
// Contents of header file "getInfo.h":
getInfo(char *detail);
```

The proxy of the service is used by a client to request a piece of information and the service responds with:

```
HTTP/1.1 200 OK
Content-Type:  text/xml
Content-Length:  nnn

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
<SOAP-ENV:Body>
<getInfoResponse>
<detail>
<picture>Mona Lisa by <i>Leonardo da Vinci</i></picture>
</detail>
</getInfoResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

As a result of the mixed content decoding, the `detail` string contains "`<picture>Mona Lisa by <i>Leonardo da Vinci</i></picture>`".

11.3.6 C++ Strings

gSOAP supports C++ strings `std::string` and `std::wstring` wide character strings. For example:

```
typedef std::string xsd__string;
class ns__myClass
{ public:
  xsd__string s; // serialized with xsi:type="xsd:string"
  std::string t; // serialized without xsi:type
...
};
```

Caution: Please avoid mixing `std::string` and C strings (**char***) in the header file when using SOAP 1.1 encoding. The problem is that multi-referenced strings in SOAP encoded messages cannot be assigned simultaneously to a `std::string` and a **char*** string.

11.3.7 Changing the Encoding Precision of float and double Types

The `double` encoding format is by default set to "%.18G" (see a manual on `printf` text formatting in C), i.e. at most 18 digits of precision to limit a loss in accuracy. The `float` encoding format is by default "%.9G", i.e. at most 9 digits of precision.

The encoding format of a double type can be set by assigning a format string to `soap.double_format`, where `soap` is a variable that contains the current runtime context. For example:

```
struct soap soap;
soap_init(&soap); // sets double_format = "%.18G"
soap.double_format = "%e"; // redefine
```

which causes all doubles to be encoded in scientific notation. Likewise, the encoding format of a float type can be set by assigning a format string to the static soap_float_format string variable. For example:

```
struct soap soap;
soap_init(&soap); // sets float_format = "%.9G"
soap.float_format = "%.4f"; // redefine
```

which causes all floats to be encoded with four digits precision.

Caution: The format strings are not automatically reset before or after SOAP communications. An error in the format string may result in the incorrect encoding of floating point values.

A special case for C format string patterns is introduced in gSOAP 2.8.18. A C format string that is used as a pattern for a typedef float or double in the gSOAP header file is used to format the output of the floating point value in XML. For example:

```
typedef float time__ratio "%5.2f";
```

This will output the float with 5 digits total and 2 digits after the decimal point.

When `xs:totalDigits` and `xs:fractionDigits` are given in a XSD file, then also a C format string is produced to output floating point values with the proper precision and scale. For example:

```
<simpleType name="ratio">
  <restriction base="xsd:float">
    <totalDigits value="5"/>
    <fractionDigits value="2"/>
  </restriction>
</simpleType>
```

11.3.8   INF, -INF, and NaN Values of float and double Types

The gSOAP runtime stdsoap2.cpp and header file stdsoap2.h support the marshalling of IEEE INF, -INF, and NaN representations. Under certain circumstances this may break if the hardware and/or C/C++ compiler does not support these representations. To remove the representations, remove the inclusion of the <math.h> header file from the stdsoap2.h file. You can control the representations as well, which are defined by the macros:

```
#define FLT_NAN
#define FLT_PINFTY
#define FLT_NINFTY
#define DBL_NAN
#define DBL_PINFTY
#define DBL_NINFTY
```

11.4 Enumeration Serialization

Enumerations are generally useful for the declaration of named integer-valued constants, also called enumeration constants.

11.4.1 Serialization of Symbolic Enumeration Constants

The gSOAP soapcpp2 tool encodes the constants of enumeration-typed variables in symbolic form using the names of the constants when possible to comply to SOAP's enumeration encoding style. Consider for example the following enumeration of weekdays:

enum weekday {Mon, Tue, Wed, Thu, Fri, Sat, Sun};

The enumeration-constant Mon, for example, is encoded as

```
<weekday xsi:type="weekday">Mon</weekday>
```

The value of the `xsi:type` attribute is the enumeration-type identifier's name. If the element is independent as in the example above, the element name is the enumeration-type identifier's name.

The encoding of complex types such as enumerations requires a reference to an XML Schema through the use of a namespace prefix. The namespace prefix can be specified as part of the enumeration-type identifier's name, with the usual namespace prefix conventions for identifiers. This can be used to explicitly specify the encoding style. For example:

enum ns1__weekday {Mon, Tue, Wed, Thu, Fri, Sat, Sun};

The enumeration-constant Sat, for example, is encoded as:

```
<ns1:weekday xsi:type="ns1:weekday">Sat</ns1:weekday>
```

The corresponding XML Schema for this enumeration data type would be:

```
<xsd:element name="weekday" type="tns:weekday"/>
<xsd:simpleType name="weekday">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Mon"/>
    <xsd:enumeration value="Tue"/>
    <xsd:enumeration value="Wed"/>
    <xsd:enumeration value="Thu"/>
    <xsd:enumeration value="Fri"/>
    <xsd:enumeration value="Sat"/>
    <xsd:enumeration value="Sun"/>
  </xsd:restriction>
</xsd:simpleType>
```

11.4.2 Encoding of Enumeration Constants

If the value of an enumeration-typed variable has no corresponding named constant, the value is encoded as a signed integer literal. For example, the following declaration of a workday enumeration type lacks named constants for Saturday and Sunday:

    enum ns1__workday {Mon, Tue, Wed, Thu, Fri};

If the constant 5 (Saturday) or 6 (Sunday) is assigned to a variable of the workday enumeration type, the variable will be encoded with the integer literals 5 and 6, respectively. For example:

    <ns1:workday xsi:type="ns1:workday">5</ns1:workday>

Since this is legal in C++ and SOAP allows enumeration constants to be integer literals, this method ensures that non-symbolic enumeration constants are correctly communicated to another party if the other party accepts literal enumeration constants (as with the gSOAP soapcpp2 tool).

Both symbolic and literal enumeration constants can be decoded.

To enforce the literal enumeration constant encoding and to get the literal constants in the WSDL file, use the following trick:

    enum ns1__nums { _1 = 1, _2 = 2, _3 = 3 };

The difference with an enumeration type without a list of values and the enumeration type above is that the enumeration constants will appear in the WSDL service description.

11.4.3 Initialized Enumeration Constants

The gSOAP soapcpp2 compiler supports the initialization of enumeration constants, as in:

    enum ns1__relation {LESS = -1, EQUAL = 0, GREATER = 1};

The symbolic names LESS, EQUAL, and GREATER will appear in the SOAP payload for the encoding of the ns1__relation enumeration values -1, 0, and 1, respectively.

11.4.4 How to "Reuse" Symbolic Enumeration Constants

A well-known deficiency of C and C++ enumeration types is the lack of support for the reuse of symbolic names by multiple enumerations. That is, the names of all the symbolic constants defined by an enumeration cannot be reused by another enumeration. To force encoding of the same symbolic name by different enumerations, the identifier of the symbolic name can end in an underscore (_) or any number of underscores to distinguish it from other symbolic names in C++. This guarantees that the SOAP encoding will use the same name, while the symbolic names can be distinguished in C++. Effectively, the underscores are removed from a symbolic name prior to encoding.

Consider for example:

```
enum ns1__workday {Mon, Tue, Wed, Thu, Fri};
enum ns1__weekday {Mon_, Tue_, Wed_, Thu_, Fri_, Sat_, Sun_};
```

which will result in the encoding of the constants of enum ns1__weekday without the underscore, for example as Mon.

As an alternative to the trailing underscores that can get quite long for commonly used symbolic enum names, you can use the following convention with double underscores to add the enum name to the enum constants:

```
enum prefixedname { prefixedname__enumconst1, prefixedname__enumconst2, ... };
```

where the type name of the enumeration prefixedname is a prefixed name, such as:

```
enum ns1__workday { ns1__workday__Mon, ns1__workday__Tue, ns1__workday__Wed, ns1__workday__Thu,
ns1__workday__Fri };
enum ns1__weekday { ns1__workday__Mon, ns1__workday__Tue, ns1__workday__Wed, ns1__workday__Thu,
ns1__workday__Fri, ns1__workday__Sat, ns1__workday__Sun };
```

This ensures that the XML schema enumeration values are still simply Mon, Tue, Wed, Thu, Fri, Sat, and Sun.

Caution: The following declaration:

```
enum ns1__workday {Mon, Tue, Wed, Thu, Fri};
enum ns1__weekday {Sat = 5, Sun = 6};
```

will not properly encode the weekday enumeration when you assume that workdays are part of weekdays, because it lacks the named constants for workday in its enumeration list. All enumerations must be self-contained and cannot use enum constants of other enumerations.

11.4.5 Boolean Enumeration Serialization for C

When developing a C Web service application, the C++ bool type should not be used since it is not usually supported by the C compiler. Instead, an enumeration type should be used to serialize true/false values as xsd:boolean Schema type enumeration values. The xsd:boolean XML Schema type is defined as:

```
enum xsd__boolean {false_, true_};
```

The value false_, for example, is encoded as:

```
<xsd:boolean xsi:type="xsd:boolean">false</xsd:boolean>
```

Peculiar of the SOAP boolean type encoding is that it only defines the values 0 and 1, while the built-in XML Schema boolean type also defines the false and true symbolic constants as valid values. The following example declaration of an enumeration type lacks named constants altogether to force encoding of the enumeration values as literal constants:

```
enum SOAP_ENC__boolean {};
```

The value 0, for example, is encoded with an integer literal:

```
<SOAP-ENC:boolean xsi:type="SOAP-ENC:boolean">0<SOAP-ENC:boolean>
```

11.4.6   Bitmask Enumeration Serialization

A bitmask is an enumeration of flags such as declared with C#'s [Flags] **enum** annotation. gSOAP supports bitmask encoding and decoding for interoperability. However, bitmask types are not standardized with SOAP RPC.

A special syntactic convention is used in the header file input to the gSOAP soapcpp2 compiler to indicate the use of bitmasks with an asterisk:

> **enum** * *name* { *enum-constant, enum-constant, ...* };

The gSOAP soapcpp2 compiler will encode the enumeration constants as flags, i.e. as a series of powers of 2 starting with 1. The enumeration constants can be or-ed to form a bitvector (bitmask) which is encoded and decoded as a list of symbolic values in SOAP. For example:

> **enum** * ns__machineStatus { ON, BELT, VALVE, HATCH};
> **int** ns__getMachineStatus(**char** *name, **char** *enum ns__machineStatus result);

Note that the use of the **enum** does not require the asterisk, only the definition. The gSOAP soapcpp2 compiler generates the enumeration:

> **enum** ns__machineStatus { ON=1, BELT=2, VALVE=4, HATCH=8};

A service operation implementation in a Web service can return:

> **int** ns__getMachineStatus(**struct** soap *soap, **char** *name, **enum** ns__machineStatus result)
> { ...
>   *result = BELT — HATCH;
>   **return** SOAP_OK;
> }

11.5   Struct Serialization

A **struct** data type is encoded as an XML Schema complexType (SOAP-encoded compound data type) such that the **struct** name forms the data type's element name and schema type and the fields of the **struct** are the data type's accessors. This encoding is identical to the **class** instance encoding without inheritance and method declarations, see Section 11.6 for further details. However, the encoding and decoding of **struct**s is more efficient compared to **class** instances due to the lack of inheritance and the requirement by the serialization routines to check inheritance properties at run time.

Certain type of fields of a **struct** can be (de)serialized as XML attributes using the @ type qualifier. See Section 11.6.7 for more details.

See Section 10.3 for more details on the struct/class member field serialization and the resulting element and attribute qualified forms.

11.6 Class Instance Serialization

A **class** instance is serialized as an XML Schema complexType (SOAP-encoded compound data type) such that the **class** name forms the data type's element name and schema type and the data member fields are the data type's accessors. Only the data member fields are encoded in the SOAP payload. Class methods are not encoded.

The general form of a **class** declaration is:

```
class [namespace_prefix__]class_name1 [:[public:] [private:] [protected:] [namespace_prefix__]class_name2]
{
   [public:] [private:] [protected:]
   field1;
   field2;
   ...
   [public:] [private:] [protected:]
   method1;
   method2;
   ...
};
```

where

namespace_prefix__ is the optional namespace prefix of the compound data type (see identifier translation rules 10.3)

class_name1 is the element name of the compound data type (see identifier translation rules 10.3).

class_name2 is an optional base class.

field is a field declaration (data member). A field MAY be declared **static** and **const** and MAY be initialized.

method is a method declaration. A method MAY be declared **virtual**, but abstract methods are not allowed. The method parameter declarations are REQUIRED to have parameter identifier names.

[**public:**] [**private:**] [**protected:**] are OPTIONAL. Only members with **public** acces permission can be serialized.

A class name is REQUIRED to be unique and cannot have the same name as a **struct**, **enum**, or service operation name specified in the header file input to the gSOAP soapcpp2 compiler. The

143

reason is that service operation requests are encoded similarly to class instances in SOAP and they are in principle undistinguishable (the method parameters are encoded just as the fields of a **class**).

Only single inheritance is supported by the gSOAP soapcpp2 compiler. Multiple inheritance is not supported, because of the limitations of the SOAP protocol.

If a constructor method is present, there MUST also be a constructor declaration with empty parameter list.

Classes should be declared "volatile" if you don't want gSOAP to add serialization methods to these classes, see Section 19.4 for more details.

Class templates are not supported by the gSOAP soapcpp2 compiler, but you can use STL containers, see Section 11.11.8. You can also define your own containers similar to STL containers.

Certain type of fields of a **struct** can be (de)serialized as XML attributes using the @ type qualifier. See Section 11.6.7 for more details.

See Section 10.3 for more details on the struct/class member field serialization and the resulting element and attribute qualified forms.

Arrays may be embedded within a class (and struct) using a pointer field and size information, see Section 11.11.7. This defines what is sometimes referred to in SOAP as "generics".

Void pointers may be used in a class (or struct), but you have to add a type field so the gSOAP runtime can determine the type of object pointed to, see Section 11.9.

A **class** instance is encoded as:

```
<[namespace-prefix:]class-name xsi:type="[namespace-prefix:]class-name">
<basefield-name1 xsi:type="...">...</basefield-name1>
<basefield-name2 xsi:type="...">...</basefield-name2>
...
<field-name1 xsi:type="...">...</field-name1>
<field-name2 xsi:type="...">...</field-name2>
...
</[namespace-prefix:]class-name>
```

where the `field-name` accessors have element-name representations of the class fields and the `basefield-name` accessors have element-name representations of the base class fields. (The optional parts resulting from the specification are shown enclosed in [].)

The decoding of a class instance allows any ordering of the accessors in the SOAP payload. However, if a base class field name is identical to a derived class field name because the field is overloaded, the base class field name MUST precede the derived class field name in the SOAP payload for decoding. gSOAP guarantees this, but interoperability with other SOAP implementations is cannot be guaranteed.

11.6.1   Example

The following example declares a base class ns__Object and a derived class ns__Shape:

```
// Contents of file "shape.h":
class ns__Object
```

```
{
  public:
  char *name;
};
class ns__Shape : public ns__Object
{
  public:
  int sides;
  enum ns__Color {Red, Green, Blue} color;
  ns__Shape();
  ns__Shape(int sides, enum ns__Green color);
  ~ns__Shape();
};
```

The implementation of the methods of **class** ns__Shape must not be part of the header file and need to be defined elsewhere.

An instance of **class** ns__Shape with name Triangle, 3 sides, and color Green is encoded as:

```
<ns:Shape xsi:type="ns:Shape">
<name xsi:type="string">Triangle</name>
<sides xsi:type="int">3</sides>
<color xsi:type="ns:Color">Green</color>
</ns:shape>
```

The namespace URI of the namespace prefix `ns` must be defined by a namespace mapping table, see Section 10.4.

11.6.2 Initialized static const Fields

A data member field of a class declared as **static const** is initialized with a constant value at compile time. This field is encoded in the serialization process, but is not decoded in the deserialization process. For example:

```
// Contents of file "triangle.h":
class ns__Triangle : public ns__Object
{
  public:
  int size;
  static const int sides = 3;
};
```

An instance of **class** ns__Triangle is encoded in SOAP as:

```
<ns:Triangle xsi:type="ns:Triangle">
<name xsi:type="string">Triangle</name>
<size xsi:type="int">15</size>
<sides xsi:type="int">3>/sides>
</ns:Triangle>
```

Decoding will ignore the sides field's value.

Caution: The current gSOAP implementation does not support encoding **static const** fields, due to C++ compiler compatibility differences. This feature may be provided the future.

11.6.3 Class Methods

A **class** declaration in the header file input to the gSOAP soapcpp2 compiler MAY include method declarations. The method implementations MUST NOT be part of the header file but are required to be defined in another C++ source that is externally linked with the application. This convention is also used for the constructors and destructors of the **class**.

Dynamic binding is supported, so a method MAY be declared **virtual**.

11.6.4 Getter and Setter Methods

Setter and getter methods are invoked at run time upon serialization and deserialization of class instances, respectively. The use of setter and getter methods adds more flexibility to the serialization and deserialization process.

A setter method is called in the serialization phase from the virtual soap_serialization method generated by the gSOAP soapcpp2 compiler. You can use setter methods to process a class instance just before it is serialized. A setter method can be used to convert application data, such as translating transient application data into serializable data, for example. You can also use setter methods to retrieve dynamic content and use it to update a class instance right before serialization. Remember setters as "set to serialize" operations.

Getter methods are invoked after deserialization of the instance. You can use them to adjust the contents of class instances after all their members have been deserialized. Getters can be used to convert deserialized members into transient members and even invoke methods to process the deserialized data on the fly.

Getter and setter methods have the following signature:

```
[virtual] int get(struct soap *soap) [const];
[virtual] int set(struct soap *soap);
```

The active soap struct will be passed to the get and set methods. The methods should return SOAP_OK when successful. A setter method should prepare the contents of the class instance for serialization. A getter method should process the instance after deserialization.

Here is an example of a base64 binary class:

```
class xsd__base64Binary
{ public:
  unsignedchar *__ptr;
  int __size;
  int get(struct soap *soap);
  int set(struct soap *soap);
};
```

Suppose that the type and options members of the attachment should be set when the class is about to be serialized. This can be accomplished with the `set` method from the information provided by the `__ptr` to the data and the soap struct passed to the `set` method (you can pass data via the **void***soap.user field).

The `get` method is invoked after the base64 data has been processed. You can use it for post-processing purposes.

Here is another example. It defines a primitive `update` type. The class is a wrapper for the `time_t` type, see Section 11.3.2. Therefore, elements of this type contain `xsd:dateType` data.

```
class update
{ public:
  time_t __item;
  int set(struct soap *soap);
};
```

The setter method assigns the current time:

```
int update::set(struct soap *soap)
{
  this->__item = time(NULL);
  return SOAP_OK;
}
```

Therefore, serialization results in the inclusion of a time stamp in XML.

Caution: a `get` method is invoked only when the XML element with its data was completely parsed. The method is not invoked when the element is an `xsi:nil` element or has an `href` attribute.

Caution: The `soap_serialize` method of a class calls the setter (when provided). However, the `soap_serialize` method is declared **const** while the setter should be allowed to modify the contents of the class instance. Therefore, the gSOAP-generated code recasts the instance and the **const** is removed when invoking the setter.

11.6.5   Streaming XML with Getter and Setter Methods

Getter methods enable streaming XML operations. A getter method is invoked when the object is deserialized and the rest of the SOAP/XML message has not been processed yet. For example, you can add a getter method to the SOAP Header class to implement header processing logic that is activated as soon as the SOAP Header is received. An example code is shown below:

```
class h__Authentication
{ public:
  char *id;
  int get(struct soap *soap);
};
class SOAP_ENV__Header
{ public:
  h__Authentication *h__authentication;
};
```

The Authentication SOAP Header field is instantiated and decoded. After decoding, the getter method is invoked, which can be used to check the id before the rest of the SOAP message is processed.

11.6.6 Polymorphism, Derived Classes, and Dynamic Binding

Interoperability between client and service applications developed with gSOAP is established even when clients and/or services use derived classes instead of the base classes used in the declaration of the service operation parameters. A client application MAY use pointers to instances of derived classes for the input parameters of a service operation. If the service was compiled with a declaration and implementation of the derived class, the service operation base class input parameters are demarshalled and a derived class instance is created instead of a base class instance. If the service did not include a declaration of the derived class, the derived class fields are ignored and a base class instance is created. Therefore, interoperability is guaranteed even when the client sends an instance of a derived classes and when a service returns an instance of a derived class.

The following example declares Base and Derived classes and a service operation that takes a pointer to a Base class instance and returns a Base class instance:

```
// Contents of file "derived.h"
class Base
{
  public:
  char *name;
  Base();
  virtual void print();
};
class Derived : public Base
{
  public:
  int num;
  Derived();
  virtual void print();
};
int method(Base *in, struct methodResponse { Base *out; } &result);
```

This header file specification is processed by the gSOAP soapcpp2 compiler to produce the stub and skeleton routines which are used to implement a client and service. The pointer of the service operation is also allowed to point to Derived class instances and these instances will be marshalled as Derived class instances and send to a service, which is in accord to the usual semantics of parameter passing in C++ with dynamic binding.

The Base and Derived class method implementations are:

```
// Method implementations of the Base and Derived classes:
#include "soapH.h"
...
Base::Base()
{
```

```
    cout << "created a Base class instance" << endl;
}
Derived::Derived()
{
    cout << "created a Derived class instance" << endl;
}
Base::print()
{
    cout << "print(): Base class instance " << name << endl;
}
Derived::print()
{
    cout << "print(): Derived class instance " << name << " " << num << endl;
}
```

Below is an example CLIENT application that creates a Derived class instance that is passed as the input parameter of the service operation:

```
// CLIENT
#include "soapH.h"
int main()
{
    struct soap soap;
    soap_init(&soap);
    Derived obj1;
    Base *obj2;
    struct methodResponse r;
    obj1.name = "X";
    obj1.num = 3;
    soap_call_method(&soap, url, action, &obj1, r);
    r.obj2->print();
}
...
```

The following example SERVER1 application copies a class instance (Base or Derived class) from the input to the output parameter:

```
// SERVER1
#include "soapH.h"
int main()
{
    soap_serve(soap_new());
}
int method(struct soap *soap, Base *obj1, struct methodResponse &result)
{
    obj1->print();
    result.obj2 = obj1;
    return SOAP_OK;
}
...
```

The following messages are produced by the CLIENT and SERVER1 applications:

CLIENT: created a Derived class instance
SERVER1: created a Derived class instance
SERVER1: print(): Derived class instance X 3
CLIENT: created a Derived class instance
CLIENT: print(): Derived class instance X 3

Which indicates that the derived class kept its identity when it passed through SERVER1. Note that instances are created both by the CLIENT and SERVER1 by the demarshalling process.

Now suppose a service application is developed that only accepts Base class instances. The header file is:

```
// Contents of file "base.h":
class Base
{
  public:
  char *name;
  Base();
  virtual void print();
};
int method(Base *in, Base *out);
```

This header file specification is processed by the gSOAP soapcpp2 tool to produce skeleton routine which is used to implement a service (so the client will still use the derived classes).

The method implementation of the Base class are:

```
// Method implementations of the Base class:
#include "soapH.h"
...
Base::Base()
{
  cout << "created a Base class instance" << endl;
}
Base::print()
{
  cout << "print(): Base class instance " << name << endl;
}
```

And the SERVER2 application is that uses the Base class is:

```
// SERVER2
#include "soapH.h"
int main()
{
  soap_serve(soap_new());
}
int method(struct soap *soap, Base *obj1, struct methodResponse &result)
```

150

```
{
  obj1–>print();
  result.obj2 = obj1;
  return SOAP_OK;
}
...
```

Here are the messages produced by the CLIENT and SERVER2 applications:

CLIENT: created a Derived class instance
SERVER2: created a Base class instance
SERVER2: print(): Base class instance X
CLIENT: created a Base class instance
CLIENT: print(): Base class instance X

In this example, the object was passed as a Derived class instance to SERVER2. Since SERVER2 only implements the Base class, this object is converted to a Base class instance and send back to CLIENT.

11.6.7 XML Attributes

The SOAP RPC/LIT and SOAP DOC/LIT encoding styles support XML attributes in SOAP messages while SOAP RPC with "Section 5" encoding does not support XML attributes other than the SOAP and XSD specific attributes. SOAP RPC "Section 5" encoding has advantages for cross-language interoperability and data encodings such as graph serialization. However, RPC/LIT and DOC/LIT enables direct exchange of XML documents, which may include encoded application data structures. Language interoperability is compromised, because no mapping between XML and the typical language data types is defined. The meaning of the RPC/LIT and DOC/LIT XML content is Schema driven rather than application/language driven.

gSOAP supports XML attribute (de)serialization of members in structs and classes. Attributes are primitive XSD types, such as strings, enumerations, boolean, and numeric types. To declare an XML attribute in a struct/class, the qualifier @ is used with the type of the attribute. The type must be primitive type (including enumerations and strings), which can be declared with or without a **typedef** to associate a XSD type with the C/C+ type. For example

```
typedef char *xsd__string;
typedef bool *xsd__boolean;
enum ns__state { _0, _1, _2 };
struct ns__myStruct
{
  @ xsd__string ns__type; // encode as XML attribute 'ns:type' of type 'xsd:string'
  @ xsd__boolean ns__flag = false; // encode as XML attribute 'ns:flag' of type 'xsd:boolean'
  @ enum ns__state ns__state = _2; // encode as XML attribute 'ns:state' of type 'ns:state'
  struct ns__myStruct *next;
};
```

The @ qualifier indicates XML attribute encoding for the ns__type, ns__flag, and ns__state fields. Note that the namespace prefix ns is used to distinguish these attributes from any other attributes such as xsi:type (ns:type is not to be confused with xsi:type).

Default values can be associated with any field that has a primitive type in a struct/class, as is illustrated in this example. The default values are used when the receiving message does not contain the corresponding values.

String attributes are optional. Other type of attributes should be declared as pointers to make them optional:

```
struct ns__myStruct
{
  @int *a; // omitted when NULL
};
```

Because a service operation request and response is essentially a struct, XML attributes can also be associated with method requests and responses. For example

```
int ns__myMethod(@char *ns__name, ...);
```

Attributes can also be attached to the dynamic arrays, binary types, and wrapper classes/structs of primitive types. Wrapper classes are described in Section 11.3.2. For example

```
struct xsd__string
{
  char *__item;
  @ xsd__boolean flag;
};
```

and

```
struct xsd__base64Binary
{
  unsigned char *__ptr;
  int __size;
  @ xsd__boolean flag;
};
```

The attribute declarations MUST follow the __item, __ptr, and __size fields which define the characteristics of wrapper structs/classes and dynamic arrays.

Caution: Do not use XML attributes with SOAP RPC encoding. You can only use attributes with RPC literal encoding.

11.6.8 QName Attributes and Elements

gSOAP ensures the proper decoding of XSD QNames. An element or attribute with type QName (Qualified Name) contains a namespace prefix and a local name. You can declare a QName type as a **typedef char** *xsd__QName. Values of type QName are internally handled as regular strings. gSOAP takes care of the proper namespace prefix mappings when deserializing QName values. For example

152

```
typedef char *xsd__QName;
struct ns__myStruct
{
   xsd__QName elt = "ns:xyz"; // QName element with default value "ns:xyz"
   @ xsd__QName att = "ns:abc"; // QName attribute with default value "ns:abc"
};
```

When the elt and att fields are serialized, their string contents are just transmitted (which means that the application is responsible to ensure proper formatting of the QName strings prior to transmission). When the fields are deserialized however, gSOAP takes care mapping the qualifiers to the appropriate namespace prefixes. Suppose that the inbound value for the elt is x:def, where the namespace name associated with the prefix x matches the namespace name of the prefix ns (as defined in the namespace mapping table). Then, the value is automatically converted into ns:def. If the namespace name is not in the table, then x:def is converted to "URI":def where "URI" is the namespace URI bound to x in the message received. This enables an application to retrieve the namespace information, whether it is in the namespace mapping table or not.

Note: QName is a pre-defined typedef type and used by gSOAP to (de)serialize SOAP Fault codes which are QName elements.

11.7   Union Serialization

A **union** is only serialized if the **union** is used within a **struct** or **class** declaration that includes a **int** __union field that acts as a *discriminant* or *selector* for the **union** fields. The selector stores run-time usage information about the **union** fields. That is, the selector is used to enumerate the **union** fields such that the gSOAP engine is able to select the correct **union** field to serialize.

A **union** within a **struct** or **class** with a selector field represents xs:choice within a Schema complexType component. For example:

```
struct ns__PO
{ ... };
struct ns__Invoice
{ ... };
union ns__PO_or_Invoice
{
   struct ns__PO po;
   struct ns__Invoice invoice;
};
struct ns__composite
{
   char *name;
   int __union;
   union ns__PO_or_Invoice value;
};
```

The **union** ns__PO_or_Invoice is expanded as a xs:choice:

```
<complexType name="composite">
  <sequence>
```

```
        <element name="name" type="xsd:string"/>
        <choice>
          <element name="po" type="ns:PO"/>
          <element name="invoice" type="ns:Invoice"/>
        </choice>
      </sequence>
    </complexType>
```

Therefore, the name of the **union** and field can be freely chosen. However, the **union** name should be qualified (as shown in the example) to ensure instances of XML Schemas with `elementFormDefault="qualified"` are correctly serialized (`po` and `invoice` are `ns:` qualified).

The **int** __union field selector's values are determined by the `soapcpp2` compiler. Each **union** field name has a selector value formed by:

SOAP_UNION_*union-name*_*field-name*

These selector values enumerate the **union** fields starting with 1. The value 0 (or any negative value) can be assigned to omit the serialization of the **union**, but only if explicitly allowed by validation rules, which requires `minOccurs="0"` for the `xs:choice` as follows:

```
struct ns__composite
{
  char *name;
  int __union 0; // <choice minOccurs="0">
  union ns__PO_or_Invoice value;
};
```

This way we can treat the **union** as an optional data item by setting __union=0.

Since 2.7.16 it is also possible to use a '$' as a special marker to annotate a selector field that must be of type **int** and the field name is no longer relevant:

```
struct ns__composite
{
  char *name;
  $int select 0; // <choice minOccurs="0">
  union ns__PO_or_Invoice value;
};
```

The following example shows how the **struct** ns__composite instance is initialized for serialization using the above declaration:

```
struct ns__composite data;
data.name = "...";
data.select = SOAP_UNION_ns__PO_or_Invoice_po; // select PO
data.value.po.number = ...; // populate the PO
```

Note that failing to set the selector to a valid **union** field can lead to a crash of the gSOAP serializer because it will attempt to serialize an invalid **union** field.

For deserialization of **union** types, the selector will be set to one of the **union** field selector values, as determined by the XML payload. The selector will be set to 0 or -1 when no union member was deserialized, where a negative value indicates that a member was required by validation rules. Strict validation enabled with SOAP_XML_STRICT results in a validation fault.

When more than one **union** is used in a **struct** or **class**, the _ _union selectors must be renamed to avoid name clashes by using suffixes as in:

```
struct ns_ _composite
{
    char *name;
    $int sel_value; // = SOAP_UNION_ns_ _PO_or_Invoice_[po—invoice]
    union ns_ _PO_or_Invoice value;
    $int sel_data; // = SOAP_UNIO_ns_ _Email_or_Fax_[email—fax]
    union ns_ _Email_or_Fax data;
};
```

11.8 Serializing Pointer Types

The serialization of a pointer to a data type amounts to the serialization of the data type in SOAP and the SOAP encoded representation of a pointer to the data type is indistinguishable from the encoded representation of the data type pointed to.

11.8.1 Multi-Referenced Data

A data structure pointed to by more than one pointer is serialized as SOAP multi-reference data. This means that the data will be serialized only once and identified with a unique `id` attribute. The encoding of the pointers to the shared data is done through the use of `href` or `ref` attributes to refer to the multi-reference data. See Section 9.12 on options to control the serialization of multi-reference data. To turn multi-ref off, use SOAP_XML_TREE to process plain tree-based XML. To completely eliminate multi-ref (de)serialization use the WITH_NOIDREF compile-time flag to permanently disable id-href processing. Cyclic C/C++ data structures are encoded with multi-reference SOAP encoding. Consider for example the following a linked list data structure:

```
typedef char *xsd_ _string;
struct ns_ _list
{
    xsd_ _string value;
    struct ns_ _list *next;
};
```

Suppose a cyclic linked list is created. The first node contains the value "**abc**" and points to a node with value "**def**" which in turn points to the first node. This is encoded as:

```
<ns:list id="1" xsi:type="ns:list">
  <value xsi:type="xsd:string">abc</value>
  <next xsi:type="ns:list">
    <value xsi:type="xsd:string">def</value>
```

```
      <next href="#1"/>
    </next>
  </ns:list>
```

In case multi-referenced data is received that "does not fit in a pointer-based structure", the data is copied. For example, the following two **struct**s are similar, except that the first uses pointer-based fields while the other uses non-pointer-based fields:

```
typedef long xsd_ _int;
struct ns_ _record
{
  xsd_ _int *a;
  xsd_ _int *b;
} P;
struct ns_ _record
{
  xsd_ _int a;
  xsd_ _int b;
} R;
...
  P.a = &n;
  P.b = &n;
...
```

Since both a and b fields of P point to the same integer, the encoding of P is multi-reference:

```
<ns:record xsi:type="ns:record">
  <a href="#1"/>
  <b href="#1"/>
</ns:record>
<id id="1" xsi:type="xsd:int">123</id>
```

Now, the decoding of the content in the R data structure that does not use pointers to integers results in a copy of each multi-reference integer. Note that the two **struct**s resemble the same XML data type because the trailing underscore will be ignored in XML encoding and decoding.

11.8.2   NULL Pointers and Nil Elements

A NULL pointer is **not** serialized, unless the pointer itself is pointed to by another pointer (but see Section 9.12 to control the serialization of NULLs). For example:

```
struct X
{
  int *p;
  int **q;
}
```

Suppose pointer q points to pointer p and suppose p=NULL. In that case the p pointer is serialized as

```
<...  id="123" xsi:nil="true"/>
```

and the serialization of q refers to `href="#123"`. Note that SOAP 1.1 does not support pointer to pointer types (!), so this encoding is specific to gSOAP. The pointer to pointer encoding is rarely used in codes anyway. More common is a pointer to a data type such as a **struct** with pointer fields.

Caution: When the deserializer encounters an XML element that has a `xsi:nil="true"` attribute but the corresponding C++ data is not a pointer or reference, the deserializer will terminate with a SOAP_NULL fault when the SOAP_XML_STRICT flag is set. The types section of a WSDL description contains information on the "nilability" of data.

11.9   Void Pointers

In general, void pointers (**void***) cannot be (de)serialized because the type of data referred to is untyped. To enable the (de)serialization of the void pointers that are members of structs or classes, you can insert a **int** __type field right before the void pointer field. The **int** __type field contains run time information on the type of the data pointed to by **void*** member in a struct/class to enable the (de)serialization of this data. The **int** __type field is set to a SOAP_TYPE_X value, where X is the name of a type. gSOAP generates the SOAP_TYPE_X definitions in soapH.h and uses them internally to uniquely identify the type of each object. The type naming conventions outlined in Section 7.5.3 are used to determine the type name for X.

Here is an example to illustrate the (de)serialization of a **void*** field in a struct:

```
struct myStruct
{
  int __type; // the SOAP_TYPE pointed to by p
  void *p;
};
```

The __type integer can be set to 0 at run time to omit the serialization of the void pointer field.

The following example illustrates the initialization of myStruct with a void pointer to an int:

```
struct myStruct S;
int n;
S.p = &n;
S.__type = SOAP_TYPE_int;
```

The serialized output of S contains the integer.

The deserializer for myStruct will automatically set the __type field and void pointer to the deserialized data, provided that the XML content for p carries the `xsi:type` attribute from which gSOAP can determine the type.

Important: when (de)serializing strings via a **void*** field, the **void*** pointer MUST directly point to the string value rather than indirectly as with all other types. For example:

```
struct myStruct S;
S.p = (void*)"Hello";
```

157

```
    S.__type = SOAP_TYPE_string;
```

This is the case for all string-based types, including types defined with **typedef char***.

You may use an arbitrary suffix with the __type fields to handle multiple void pointers in structs/classes. For example

```
    struct myStruct
    {
      int __typeOfp; // the SOAP_TYPE pointed to by p
      void *p;
      int __typeOfq; // the SOAP_TYPE pointed to by q
      void *q;
    };
```

Because service method parameters are stored within structs, you can use __type and **void*** parameters to pass polymorphic arguments without having to define a C++ class hierarchy (Section 11.6.6). For example:

```
    typedef char *xsd__string;
    typedef int xsd__int;
    typedef float xsd__float;
    enum ns__status { on, off };
    struct ns__widget { xsd__string name; xsd__int part; }; int ns__myMethod(int __type, void *data,
    struct ns__myMethodResponse { int __type; void *return_; } *out);
```

This method has a polymorphic input parameter data and a polymorphic output parameter return_. The __type parameters can be one of SOAP_TYPE_xsd__string, SOAP_TYPE_xsd__int, SOAP_TYPE_xsd__float, SOAP_TYPE_ns__status, or SOAP_TYPE_ns__widget. The WSDL produced by the gSOAP soapcpp2 compiler declares the polymorphic parameters of type xsd:anyType which is "too loose" and doesn't allow the gSOAP importer to handle the WSDL accurately. Future gSOAP releases might replace xsd:anyType with a choice schema type that limits the choice of types to the types declared in the header file.

11.10   Fixed-Size Arrays

Fixed size arrays are encoded as per SOAP 1.1 one-dimensional array types. Multi-dimensional fixed size arrays are encoded by gSOAP as nested one-dimensional arrays in SOAP. Encoding of fixed size arrays supports partially transmitted and sparse array SOAP formats.

The decoding of (multi-dimensional) fixed-size arrays supports the SOAP multi-dimensional array format as well as partially transmitted and sparse array formats.

An example:

```
    // Contents of header file "fixed.h":
    struct Example
    {
      float a[2][3];
    };
```

This specifies a fixed-size array part of the **struct** Example. The encoding of array `a` is:

```
<a xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="float[][2]">
<SOAP-ENC:Array xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="float[3]"
<float xsi:type="float">...</float>
<float xsi:type="float">...</float>
<float xsi:type="float">...</float>
</SOAP-ENC:Array>
<SOAP-ENC:Array xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="float[3]"
<float xsi:type="float">...</float>
<float xsi:type="float">...</float>
<float xsi:type="float">...</float>
</SOAP-ENC:Array>
</a>
```

Caution: Any decoded parts of a (multi-dimensional) array that do not "fit" in the fixed size array are ignored by the deserializer.

11.11 Dynamic Arrays

As the name suggests, dynamic arrays are much more flexible than fixed-size arrays and dynamic arrays are better adaptable to the SOAP encoding and decoding rules for arrays. In addition, a typical C application allocates a dynamic array using `malloc`, assigns the location to a pointer variable, and deallocates the array later with `free`. A typical C++ application allocates a dynamic array using `new`, assigns the location to a pointer variable, and deallocates the array later with `delete`. Such dynamic allocations are flexible, but pose a problem for the serialization of data: how does the array serializer know the length of the array to be serialized given only a pointer to the sequence of elements? The application stores the size information somewhere. This information is crucial for the array serializer and has to be made explicitly known to the array serializer by packaging the pointer and array size information within a **struct** or **class**.

11.11.1 SOAP Array Bounds Limits

SOAP encoded arrays use the `SOAP-ENC:Array` type and the `SOAP-ENC:arrayType` attribute to define the array dimensionality and size. As a security measure to avoid denial of service attacks based on sending a huge array size value requiring the allocation of large chunks of memory, the total number of array elements set by the `SOAP-ENC:arrayType` attribute cannot exceed SOAP_MAXARRAYSIZE, which is set to 100,000 by default. This constant is defined in `stdsoap2.h`. This constant **only** affects multi-dimensional arrays and the dimensionality of the receiving array will be lost when the number of elements exceeds 100,000. One-dimensional arrays will be populated in sequential order as expected.

11.11.2 One-Dimensional Dynamic SOAP Arrays

A special form of **struct** or **class** is used to define one-dimensional dynamic SOAP-encoded arrays. Each array has a pointer variable and a field that records the number of elements the pointer points to in memory.

The general form of the **struct** declaration that contains a one-dimensional dynamic SOAP-encoded array is:

```
struct some_name
{
    Type *__ptr; // pointer to array of elements in memory
    int __size; // number of elements pointed to
    [[static const] int __offset [= ...];] // optional SOAP 1.1 array offset
    ... // anything that follows here will be ignored
};
```

where Type MUST be a type associated with an XML Schema or MUST be a primitive type. If these conditions are not met, a vector-like XML (de)serialization is used (see Section 11.11.7). A primitive type can be used with or without a **typedef**. If the array elements are structs or classes, then the struct/class type names should have a namespace prefix for schema association, or they should be other (nested) dynamic arrays.

An alternative to a **struct** is to use a **class** with optional methods that MUST appear after the __ptr and __size fields:

```
class some_name
{
    public:
    Type *__ptr;
    int __size;
    [[static const] int __offset [= ...];]
    method1;
    method2;
    ... // any fields that follow will be ignored
};
```

To encode the data type as an array, the name of the **struct** or **class** SHOULD NOT have a namespace prefix, otherwise the data type will be encoded and decoded as a generic vector, see Section 11.11.7.

The deserializer of a dynamic array can decode partially transmitted and/or SOAP sparse arrays, and even multi-dimensional arrays which will be collapsed into a one-dimensional array with row-major ordering.

Caution: SOAP 1.2 does not support partially transmitted arrays. So the __offset field of a dynamic array is ignored.

11.11.3   Example

The following example header file specifies the XMethods Service Listing service getAllSOAPServices service operation and an array of SOAPService data structures:

```
// Contents of file "listing.h":
class ns3__SOAPService
{
    public:
```

```
      int ID;
      char *name;
      char *owner;
      char *description;
      char *homepageURL;
      char *endpoint;
      char *SOAPAction;
      char *methodNamespaceURI;
      char *serviceStatus;
      char *methodName;
      char *dateCreated;
      char *downloadURL;
      char *wsdlURL;
      char *instructions;
      char *contactEmail;
      char *serverImplementation;
   };
   class ServiceArray
   {
      public:
      ns3__SOAPService *__ptr; // points to array elements
      int __size; // number of elements pointed to
      ServiceArray();
      ~ServiceArray();
      void print();
   };
   int ns__getAllSOAPServices(ServiceArray &return_);
```

An example client application:

```
   #include "soapH.h" ...
   // ServiceArray class method implementations:
   ServiceArray::ServiceArray()
   {
      __ptr = NULL;
      __size = 0;
   }
   ServiceArray::~ServiceArray()
   { // destruction handled by gSOAP
   }
   void ServiceArray::print()
   {
      for (int i = 0; i ¡ __size; i++)
         cout << __ptr[i].name << ": " << __ptr[i].homepage << endl;
   }
   ...
   // Request a service listing and display results:
   {
      struct soap soap;
      ServiceArray result;
      const char *endpoint = "www.xmethods.net:80/soap/servlet/rpcrouter";
```

```
    const char *action = "urn:xmethodsServicesManager#getAllSOAPServices";
    ...
    soap_init(&soap);
    soap_call_ns__getAllSOAPServices(&soap, endpoint, action, result);
    result.print();
    ...
    soap_destroy(&soap); // dealloc class instances
    soap_end(&soap); // dealloc deserialized data
    soap_done(&soap); // cleanup and detach soap struct
}
```

11.11.4   One-Dimensional Dynamic SOAP Arrays With Non-Zero Offset

The declaration of a dynamic array as described in 11.11 MAY include an **int** __offset field. When set to an integer value, the serializer of the dynamic array will use this field as the start index of the array and the SOAP array offset attribute will be used in the SOAP payload. Note that array offsets is a SOAP 1.1 specific feature which is not supported in SOAP 1.2.

For example, the following header file declares a mathematical Vector class, which is a dynamic array of floating point values with an index that starts at 1:

```
// Contents of file "vector.h":
typedef float xsd__float;
class Vector
{
  xsd__float *__ptr;
  int __size;
  int __offset;
  Vector();
  Vector(int n);
  float& operator[](int i);
}
```

The implementations of the Vector methods are:

```
Vector::Vector()
{
  __ptr = NULL;
  __size = 0;
  __offset = 1;
}
Vector::Vector(int n)
{
  __ptr = (float*)malloc(n*sizeof(float));
  __size = n;
  __offset = 1;
}
Vector::~Vector()
{
  if (__ptr)
```

```
      free(__ptr);
   }
   float& Vector::operator[](int i)
   {
      return __ptr[i-__offset];
   }
```

An example program fragment that serializes a vector of 3 elements:

```
   struct soap soap;
   soap_init(&soap);
   Vector v(3);
   v[1] = 1.0;
   v[2] = 2.0;
   v[3] = 3.0;
   soap_begin(&soap);
   v.serialize(&soap);
   v.put("vec");
   soap_end(&soap);
```

The output is a partially transmitted array:

```
<vec xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:float[4]" SOAP-ENC:offset="[1]">
<item xsi:type="xsd:float">1.0</item>
<item xsi:type="xsd:float">2.0</item>
<item xsi:type="xsd:float">3.0</item>
</vec>
```

Note that the size of the encoded array is necessarily set to 4 and that the encoding omits the non-existent element at index 0.

The decoding of a dynamic array with an __offset field is more efficient than decoding a dynamic array without an __offset field, because the __offset field will be assigned the value of the SOAP-ENC:offset attribute instead of padding the initial part of the array with default values.

11.11.5   Nested One-Dimensional Dynamic SOAP Arrays

One-dimensional dynamic arrays MAY be nested. For example, using **class** Vector declared in the previous section, **class** Matrix is declared:

```
   // Contents of file "matrix.h":
   class Matrix
   {
      public:
      Vector *__ptr;
      int __size;
      int __offset;
      Matrix();
      Matrix(int n, int m);
      ~Matrix();
```

```
        Vector& operator[](int i);
    };
```

The Matrix type is essentially an array of pointers to arrays which make up the rows of a matrix. The encoding of the two-dimensional dynamic array in SOAP will be in nested form.

11.11.6 Multi-Dimensional Dynamic SOAP Arrays

The general form of the **struct** declaration for K-dimensional $(K > 1)$ dynamic arrays is:

```
        struct some_name
        {
            Type *__ptr;
            int __size[K];
            int __offset[K];
            ... // anything that follows here will be ignored
        };
```

where Type MUST be a type associated with an XML Schema, which means that it must be a **typedef**ed type in case of a primitive type, or a **struct**/**class** name with a namespace prefix for schema association, or another dynamic array. If these conditions are not met, a generic vector XML (de)serialization is used (see Section 11.11.7).

An alternative is to use a **class** with optional methods:

```
        class some_name
        {
            public:
            Type *__ptr;
            int __size[K];
            int __offset[K];
            method1;
            method2;
            ... // any fields that follow will be ignored
        };
```

In the above, K is a constant denoting the number of dimensions of the multi-dimensional array.

To encode the data type as an array, the name of the **struct** or **class** SHOULD NOT have a namespace prefix, otherwise the data type will be encoded and decoded as a generic vector, see Section 11.11.7.

The deserializer of a dynamic array can decode partially transmitted multi-dimensional arrays.

For example, the following declaration specifies a matrix class:

```
        typedef double xsd__double;
        class Matrix
        {
            public:
            xsd__double *__ptr;
            int __size[2];
```

```
    int __offset[2];
};
```

In contrast to the matrix class of Section 11.11.5 that defined a matrix as an array of pointers to matrix rows, this class has one pointer to a matrix stored in row-major order. The size of the matrix is determined by the __size field: __size[0] holds the number of rows and __size[1] holds the number of columns of the matrix. Likewise, __offset[0] is the row offset and __offset[1] is the columns offset.

11.11.7   Encoding XML Generics Containing Dynamic Arrays

The XML "generics" concept discussed in the SOAP encoding protocols extends the concept of a SOAP struct by allowing repetitions of elements within the struct. This is just a form of a repetition of XML elements without the SOAP-encoded array requirements. While SOAP-encoded arrays are more expressive (offset information to encode sparse arrays for example), simple repetitions of values are used more frequently.

A simple generic reperition is an array-like data structure with a repetition of an element. To achieve this, declare a dynamic array as a **struct** or **class** with a name that is qualified with a namespace prefix. SOAP arrays are declared without prefix.

For example, we define a Map structure that contains a sequence of pairs of key-val:

```
struct ns__Map
{
  int __size; // number of pairs
  struct ns__Binding {char *key; char *val;} *pair;
};
```

Since 2.7.16 it is also possible to use a '$' as a special marker to annotate a size field that must be of type **int** or size_t and the field name is no longer relevant:

```
struct ns__Map
{
  $int length; // number of pairs
  struct ns__Binding {char *key; char *val;} *pair;
};
```

This declares a dynamic array pointed to by pair and size __size. The array will be serialized and deserialized as a sequence of pairs:

```
<ns:Map xsi:type="ns:Map">
<pair xsi:type="ns:Binding">
<key>Joe</key>
<val>555 77 1234</val>
</pair>
<pair xsi:type="ns:Binding">
<key>Susan</key>
<val>555 12 6725</val>
```

```
</pair>
<pair xsi:type="ns:Binding">
<key>Pete</key>
<val>555 99 4321</val>
</pair>
</ns:Map>
```

Deserialization is less efficient compared to a SOAP-encoded array, because the size of the sequence is not part of the SOAP encoding. Internal buffering is used by the deserializer to collect the elements. When the end of the list is reached, the buffered elements are copied to a newly allocated space on the heap for the dynamic array.

Multiple arrays can be used in a struct/class to support the concept of "generics". Each array results in a repetition of elements in the struct/class. This is achieved with a **int** __size (or **$int**) field in the struct/class where the next field (i.e. below the __size field) is a pointer type. The pointer type is assumed to point to an array of values at run time. The __size field holds the number of values at run time. Multiple arrays can be embedded in a struct/class with __size fields that have a distinct names. To make the __size fields distinct, you can end them with a unique name suffix such as __sizeOfstrings, for example.

The general convention for embedding arrays is:

```
struct ns__SomeStruct
{
  ...
  int __sizename1; // number of elements pointed to
  Type1 *field1; // by this field
  ...
  int __sizename2; // number of elements pointed to
  Type2 *field2; // by this field
  ...
};
```

where name1 and name2 are identifiers used as a suffix to distinguish the __size field. These names can be arbitrary and are not visible in XML.

In 2.7.16 and higher this is simplified with a '$' marker:

```
struct ns__SomeStruct
{
  ...
  $int name1; // number of elements pointed to
  Type1 *field1; // by this field
  ...
  $int name2; // number of elements pointed to
  Type2 *field2; // by this field
  ...
};
```

For example, the following struct has two embedded arrays:

166

```
struct ns__Contact
{
  char *firstName;
  char *lastName;
  $intnPhones; // number of Phones
  ULONG64 *phoneNumber; // array of phone numbers
  $intnEmails; // number of emails
  char **emailAddress; // array of email addresses
  char *socSecNumber;
};
```

The XML serialization of an example ns__Contact is:

```
<mycontact xsi:type="ns:Contact">
  <firstName>Joe</firstName>
  <lastName>Smith</lastName>
  <phoneNumber>5551112222</phoneNumber>
  <phoneNumber>5551234567</phoneNumber>
  <phoneNumber>5552348901</phoneNumber>
  <emailAddress>Joe.Smith@mail.com</emailAddress>
  <emailAddress>Joe@Smith.com</emailAddress>
  <socSecNumber>999999999</socSecNumber>
</mycontact>
```

11.11.8 STL Containers

gSOAP supports the STL containers std::deque, std::list, std::set, and std::vector.

STL containers can only be used within classes to declare members that contain multiple values. This is somewhat similar to the embedding of arrays in structs in C as explained in Section 11.11.7, but the STL container approach is more flexible.

You need to import stldeque.h, stllist.h, stlset.h, or stlvector.h to enable std::deque, std::list, std::set, and std::vector (de)serialization. Here is an example:

```
#import "stlvector.h"
class ns__myClass
{ public:
  std::vector<int> *number;
  std::vector<xsd__string> *name;
  ...
};
```

The use of pointer members is not required but advised. The reason is that interoperability with other SOAP toolkits may lead to copying of ns__myClass instances at run time when (de)serializing multi-referenced data. When a copy is made, certain parts of the containers will be shared between the copies which could lead to disaster when the classes with their containers are deallocated. Another way to avoid this is to declare class ns__myClass within other data types via a pointer. (Interoperability between gSOAP clients and services does not lead to copying.)

The XML Schema that corresponds to the ns__myClass type is

```
<complexType name="myClass">
  <sequence>
    <element name="number" type="xsd:int" minOccurs="1" maxOccurs="unbounded"/>
    <element name="name" type="xsd:string" minOccurs="1" maxOccurs="unbounded"/>
    ...
  </sequence>
</complexType>
```

You can specify the minOccurs and maxOccurs values as explained in Section 19.2.

You can also implement your own containers similar to STL containers. The containers must be class templates and should define an iterator type, and **void** clear(), iterator begin(), iterator end(), and iterator insert(iterator pos, const_reference val). The iterator should have a dereference operator to access the container's elements. The dereference operator is used by gSOAP to send a sequence of XML element values. The insert method can be used as a setter method. gSOAP reads a sequence of XML element values and inserts them in the container via this method.

Here is in example user-defined container template class:

```
// simpleVector.h
template <class T>
class simpleVector
{
public:
  typedef T value_type;
  typedef value_type * pointer;
  typedef const value_type * const_pointer;
  typedef value_type & reference;
  typedef const value_type & const_reference;
  typedef pointer iterator;
  typedef const_pointer const_iterator;
protected:
  iterator start;
  iterator finish;
  size_t length;
public:
  simpleVector() { clear(); }
  ~simpleVector() { delete[] start; }
  void clear() { start = finish = NULL; }
  iterator begin() { return start; }
  const_iterator begin() const { return start; }
  iterator end() { return finish; }
  const_iterator end() const { return finish; }
  size_t size() const { return finish-start; }
  iterator insert(iterator pos, const_reference val)
    {
      if (!start)
        start = finish = new value_type[length = 4];
      else if (finish >= start + length)
      {
        iterator i = start;
        iterator j = new value_type[2 * length];
```

168

```
                start = j;
                finish = start + length;
                length *= 2;
                if (pos)
                    pos = j + (pos - i);
                while (i != finish)
                    *j++ = *i++;
            }
            if (pos && pos != finish)
            { iterator i = finish;
                iterator j = i - 1;
                while (j != pos)
                    *i−− = *j−−;
            }
            *finish++ = val;
            return pos;
        }
    };
```

To enable the container, we add the following two lines to our gSOAP header file:

```
    #include "simpleVector.h"
    template <class T> class simpleVector;
```

The container class should not be defined in the gSOAP header file. It must be defined in a separate header file (e.g. "simpleVector.h"). The **template** <class T> **class** simpleVector declaration ensures that gSOAP will recognize simpleVector as a container class.

Caution: when parsing XML content the container elements may not be stored in the same order given in the XML content. When gSOAP parses XML it uses the insert container methods to store elements one by one. However, element content that is "forwarded" with href attributes will be appended to the container. Forwarding can take place with multi-referenced data that is referred to from the main part of the SOAP 1.1 XML message to the independent elements that carry ids. Therefore, your application should not rely on the preservation of the order of elements in a container.

11.11.9 Polymorphic Dynamic Arrays and Lists

Polymorphic arrays (arrays of polymorphic element types) can be encoded when declared as an array of pointers to class instances. For example:

```
    class ns__Object
    {
       public:
       ...
    };
    class ns__Data: public ns__Object
    {
       public:
```

```
    ...
};
class ArrayOfObject
{
  public:
  ns__Object **__ptr; // pointer to array of pointers to Objects
  int __size; // number of Objects pointed to
  int __offset; // optional SOAP 1.1 array offset
};
class ns__Objects
{
  public:
  std::vector<ns__Object*> objects; // vector of pointers to objects
};
```

The pointers in the array can point to the ns__Object base class or ns__Data derived class instances
which will be serialized and deserialized accordingly in SOAP. That is, the array elements are
polymorphic.

Since we can't use dynamic binding to support polymorphism in C, another mechanism is available
based on the serialization of void pointers, that is, dynamic serialization of data referenced by void
pointers, see Section 11.9.

```
struct __wrapper
{
  int __type; // type T represented by SOAP_TYPE_T
  void *__item; // pointer to data of type T
};
struct ArrayOfObject
{
  struct __wrapper __ptr; // pointer to array of pointers to Objects
  int __size; // number of Objects pointed to
  int __offset; // optional SOAP 1.1 array offset
};
struct ns__Objects
{
  int __size;
  struct __wrapper *objects; // array of pointers to wrapped types
};
```

11.11.10   How to Change the Tag Names of the Elements of a SOAP Array or List

The __ptr field in a **struct** or **class** declaration of a dynamic array may have an optional suffix part
that describes the name of the tags of the SOAP array XML elements. The suffix is part of the
field name:

Type *__ptrarray_elt_name

The suffix describes the tag name to be used for all array elements. The usual identifier to XML
translations apply, see Section 10.3. The default XML element tag name for array elements is `item`
(which corresponds to the use of field name __ptritem).

Consider for example:

```
struct ArrayOfstring
{
    xsd__string *__ptrstring;    int __size; };
```

The array is serialized as:

```
<array xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[2]">
<string xsi:type="xsd:string">Hello</string>
<string xsi:type="xsd:string">World</string>
</array>
```

SOAP 1.1 and 1.2 do not require the use of a specific tag name for array elements. gSOAP will deserialize a SOAP array while ignoring the tag names. Certain XML Schemas used in doc/literal encoding may require the declaration of array element tag names.

11.12 Base64Binary XML Schema Type Encoding

The `base64Binary` XML Schema type is a special form of dynamic array declared with a pointer (__ptr) to an **unsigned char** array.

For example using a **struct**:

```
struct xsd__base64Binary
{
    unsigned char *__ptr;
    int __size;
};
```

Or with a **class**:

```
class xsd__base64Binary
{
    public:
    unsigned char *__ptr;
    int __size;
};
```

When compiled by the gSOAP `soapcpp2` tool, this header file specification will generate `base64Binary` serializers and deserializers.

The `SOAP_ENC:base64` encoding is another type for base 64 binary encoding specified by the SOAP data type schema and some SOAP applications may use this form (as indicated by their WSDL descriptions). It is declared by:

```
struct SOAP_ENC__base64
{
    unsigned char *__ptr;
    int __size;
};
```

Or with a **class**:

```
class SOAP_ENC__base64
{
    unsigned char *__ptr;
    int __size;
};
```

When compiled by the gSOAP soapcpp2 tool, this header file specification will generate `SOAP-ENC:base64` serializers and deserializers.

The advantage of using a **class** is that methods can be used to initialize and manipulate the __ptr and __size fields. The user can add methods to this class to do this. For example:

```
class xsd__base64Binary
{
    public:
    unsigned char *__ptr;
    int __size;
    xsd__base64Binary(); // Constructor
    xsd__base64Binary(struct soap *soap, int n); // Constructor
    ~xsd__base64Binary(); // Destructor
    unsigned char *location(); // returns the memory location
    int size(); // returns the number of bytes
};
```

Here are example method implementations:

```
xsd__base64Binary::xsd__base64Binary()
{
    __ptr = NULL;
    __size = 0;
}
xsd__base64Binary::xsd__base64Binary(struct soap *soap, int n)
{
    __ptr = (unsigned char*)soap_malloc(soap, n);
    __size = n;
}
xsd__base64Binary::~xsd__base64Binary()
{ }
unsigned char *xsd__base64Binary::location()
{
    return __ptr;
}
int xsd__base64Binary::size()
{
    return __size;
}
```

The following example in C/C++ reads from a raw image file and encodes the image in SOAP using the `base64Binary` type:

```
...
FILE *fd = fopen("image.jpg", "rb");
xsd__base64Binary image(&soap, filesize(fd));
fread(image.location(), image.size(), 1, fd);
fclose(fd);
soap_begin(&soap);
image.soap_serialize(&soap);
image.soap_put(&soap, "jpegimage", NULL);
soap_end(&soap);
...
```

where filesize is a function that returns the size of a file given a file descriptor.

Reading the `xsd:base64Binary` encoded image.

```
...
xsd__base64Binary image;
soap_begin(&soap);
image.get(&soap, "jpegimage");
soap_end(&soap);
...
```

The **struct** or **class** name soap_enc__base64 should be used for `SOAP-ENC:base64` schema type instead of xsd__base64Binary.

11.13   hexBinary XML Schema Type Encoding

The `hexBinary` XML Schema type is a special form of dynamic array declared with the name xsd__hexBinary and a pointer (__ptr) to an **unsigned char** array.

For example, using a **struct**:

```
struct xsd__hexBinary
{
   unsigned char *__ptr;
   int __size;
};
```

Or using a **class**:

```
class xsd__hexBinary
{
   public:
   unsigned char *__ptr;
   int __size;
};
```

When compiled by the gSOAP `soapcpp2` tool, this header file specification will generate `base64Binary` serializers and deserializers.

11.14 Literal XML Encoding Style

gSOAP supports document/literal encoding by default. Just as with SOAP RPC encoding, literal encoding requires the XML Schema of the message data to be provided e.g. in WSDL in order for the gSOAP soapcpp2 compiler to generate the (de)serialization routines. Alternatively, the optional DOM parser (dom.c and dom++.cpp) can be used to handle generic XML or arbitrary XML documents can be (de)serialized into regular C strings or wide character strings (wchar_t*) by gSOAP (see Section 11.14.1).

The //gsoap service encoding, //gsoap service method-encoding, and //gsoap service method-response-encoding directives explicitly enable SOAP encoded or literal encoded messages. For example, to enable RPC encoding style for the entire service, use:

    //gsoap ns service encoding: encoded

To enable encoding for particular service methods, use:

    //gsoap ns service method-encoding: myMethod encoded
    int ns__myMethod(...)

To enable encoding for particular service methods responses when the method request is literal, use:

    //gsoap ns service method-response-encoding: myMethod encoded
    int ns__myMethod(...)

Instead of the encoded value, you can use literal, or a specific encoding style value.

Consider the following example that uses the directive to make the literal encoding explicit. The LocalTimeByZipCode service operation of the LocalTime service provides the local time given a zip code and uses literal encoding (with MS .NET). The following header file declares the method:

    int LocalTimeByZipCode(char *ZipCode, char **LocalTimeByZipCodeResult);

Note that none of the data types need to be namespace qualified using namespace prefixes.

    //gsoap ns service name: localtime
    //gsoap ns service encoding: literal
    //gsoap ns service namespace: http://alethea.net/webservices/
    int ns__LocalTimeByZipCode(char *ZipCode, char **LocalTimeByZipCodeResult);

In this case, the method name requires to be associated with a schema through a namespace prefix, e.g. ns is used in this example. See Section 19.2 for more details on gSOAP directives. With these directives, the gSOAP soapcpp2 compiler generates client and server sources with the specified settings.

The example client program is:

    #include "soapH.h"
    #include "localtime.nsmap" // include generated map file

174

```
int main()
{
  struct soap soap;
  char *t;
  soap_init(&soap);
  if (soap_call_ns_ _LocalTimeByZipCode(&soap, "http://alethea.net/webservices/LocalTime.asmx",
"http://alethea.net/webservices/LocalTimeByZipCode", "32306", &t))
    soap_print_fault(&soap, stderr);
  else
    printf("Time = %s\n", t);
  return 0;
}
```

To illustrate the manual doc/literal setting, the following client program sets the required properties before the call:

```
#include "soapH.h"
#include "localtime.nsmap" // include generated map file
int main()
{
  struct soap soap;
  char *t;
  soap_init(&soap);
  soap.encodingStyle = NULL; // don't use SOAP encoding
  soap_set_omode(&soap, SOAP_XML_TREE);" // don't produce multi-ref data (but can accept)
  if (soap_call_ns_ _LocalTimeByZipCode(&soap, "http://alethea.net/webservices/LocalTime.asmx",
"http://alethea.net/webservices/LocalTimeByZipCode", "32306", &t))
    soap_print_fault(&soap, stderr);
  else
    printf("Time = %s\n", t);
  return 0;
}
```

The SOAP request is:

```
POST /webservices/LocalTime.asmx HTTP/1.0
Host:  alethea.net
Content-Type:  text/xml; charset=utf-8
Content-Length:  479
SOAPAction:  "http://alethea.net/webservices/LocalTimeByZipCode"

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <SOAP-ENV:Body>
    <LocalTimeByZipCode xmlns="http://alethea.net/webservices/">
<ZipCode>32306</ZipCode></LocalTimeByZipCode>
```

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

11.14.1   Serializing and Deserializing Mixed Content XML With Strings

To declare a literal XML "type" to hold XML documents in regular strings, use:

typedef char *XML;

To declare a literal XML "type" to hold XML documents in wide character strings, use:

typedef wchar_t *XML;

Note: only one of the two storage formats can be used. The differences between the use of regular strings versus wide character strings for XML documents are:

- Regular strings for XML documents MUST hold UTF-8 encoded XML documents. That is, the string MUST contain the proper UTF-8 encoding to exchange the XML document in SOAP messages.

- Wide character strings for XML documents SHOULD NOT hold UTF-8 encoded XML documents. Instead, the UTF-8 translation is done automatically by the gSOAP runtime marshalling routines.

Here is a C++ example of a service operation specification in which the parameters of the service operation uses literal XML encoding to pass an XML document to a service and back:

typedef char *XML;
ns__GetDocument(XML m__XMLDoc, XML &m__XMLDoc_);

and in C:

typedef char *XML;
ns__GetDocument(XML m__XMLDoc, XML *m__XMLDoc_);

The ns__Document is essentially a **struct** that forms the root of the XML document. The use of the underscore in the ns__Document response part of the message avoids the name clash between the **struct**s. Assuming that the namespace mapping table contains the binding of ns to `http://my.org/` and the binding of m to `http://my.org/mydoc.xsd`, the XML message is:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns="http://my.org/"
  xmlns:m="http://my.org/mydoc.xsd"
```

```
  SOAP-ENV:encodingStyle="">
  <SOAP-ENV:Body>
    <ns:GetDocument>
      <XMLDoc xmlns="http://my.org/mydoc.xsd">
        ...
      </XMLDoc>
    </ns:Document>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

When using literal encoding of method parameters and response as shown in the example above, the literal XML encoding style MUST be specified by setting soap.encodingStyle. For example, to specify no constraints on the encoding style (which is typical) use NULL:

```
struct soap soap;
soap_init(&soap);
soap.encodingStyle = NULL;
```

As a result, the `SOAP-ENV:encodingStyle` attribute will not appear in the SOAP payload.

For interoperability with Apache SOAP, use

```
struct soap soap;
soap_init(&soap);
soap.encodingStyle = "http://xml.apache.org/xml-soap/literalxml";
```

When the response parameter is an XML type, it will store the entire XML response content but without the enveloping response element.

The XML type can be used as part of any data structure to enable the rendering and parsing of custom XML documents. For example:

```
typedef char *XML;
struct ns__Data /* data in namespace 'ns' */
{
  int number;
  char *name;
  XML m__document; /* XML document in default namespace 'm' */
};
ns__Example(struct ns__Data data, struct ns__ExampleResponse { struct ns__Data data; } *out);
```

12 SOAP Fault Processing

A predeclared standard SOAP Fault data structure is generated by the gSOAP soapcpp2 tool for exchanging exception messages. The built-in **struct** SOAP_ENV__Fault data structure is defined as:

```
struct SOAP_ENV__Fault
{
  _QName faultcode; // _QName is builtin
```

```
    char *faultstring;
    char *faultactor;
    struct SOAP_ENV__Detail *detail;
    struct SOAP_ENV__Code *SOAP_ENV__Code; // MUST be a SOAP_ENV__Code struct defined
below
    char *SOAP_ENV__Reason;
    char *SOAP_ENV__Node;
    char *SOAP_ENV__Role;
    struct SOAP_ENV__Detail *SOAP_ENV__Detail; // SOAP 1.2 detail field
}; struct SOAP_ENV__Code
{
    _QName SOAP_ENV__Value;
    struct SOAP_ENV__Code *SOAP_ENV__Subcode; };
struct SOAP_ENV__Detail
{
    int __type; // The SOAP_TYPE_ of the object serialized as Fault detail
    void *fault; // pointer to the fault object, or NULL
    char *__any; // any other detail element content (stored in XML format)
};
```

The first four fields in SOAP_ENV__Fault are SOAP 1.1 specific. The last five fields are SOAP 1.2 specific. You can redefine these structures in the header file. For example, you can use a class for the SOAP_ENV__Fault and add methods for convenience.

The data structure content can be changed to the need of an application, but this is generally not necessary because the application-specific SOAP Fault details can be serialized via the __type and fault fields in the SOAP_ENV__Detail field, see Section 11.9 on the serialization of data refered to by __type and fault.

The __type field allows application data to be serialized as part of the SOAP Fault. The application data SHOULD be defined as XML elements, which requires you to declare the type names with a leading underscore to ensure that the types are compatible with XML elements and not just simpleTypes and complexTypes.

When the skeleton of a service operation returns an error (see Section 10.2), then soap.fault contains the SOAP Fault data at the receiving side (client).

Server-side faults are raised with soap_sender_fault or soap_receiver_fault. The soap_sender_fault call should be used to inform that the sender is at fault and the sender (client) should not resend the request. The soap_receiver_fault call should be used to indicate a temporary server-side problem, so a sender (client) can resend the request later. For example:

```
int ns1__myMethod(struct soap *soap, ...)
{
    ...
    return soap_receiver_fault(soap, "Resource temporarily unavailable", NULL); // return fault to
sender
}
```

In the example, the SOAP Fault details were empty (NULL). You may pass an XML fragment, which will be literally included in the SOAP Fault message. For WS-I Basic Profile compliance, you must pass an XML string with one or more namespace qualified elements, such as:

```
return soap_receiver_fault(soap, "Resource temporarily unavailable", "<errorcode xmlns='http://tempuri.org'>123</errorc
xmlns='http://tempuri.org'>abc</errorinfo>");
```

When a service operation must raise an exception with application SOAP Fault details, it does so
by assigning the soap.fault field of the current reference to the runtime context with appropriate
data associated with the exception and by returning the error SOAP_FAULT. For example:

```
soap_receiver_fault(soap, "Stack dump", NULL);
if (soap->version == 2) // SOAP 1.2 is used
{
    soap->fault->SOAP_ENV__Detail = (struct SOAP_ENV__Detail*)soap_malloc(soap, sizeof(struct
SOAP_ENV__Detail);
    soap->fault->SOAP_ENV__Detail->__type = SOAP_TYPE_ns1__myStackDataType; // stack
type
    soap->fault->SOAP_ENV__Detail->fault = sp; // point to stack
    soap->fault->SOAP_ENV__Detail->__any = NULL; // no other XML data
}
else
{
    soap->fault->detail = (struct SOAP_ENV__Detail*)soap_malloc(soap, sizeof(struct SOAP_ENV__Detail);
    soap->fault->detail->__type = SOAP_TYPE_ns1__myStackDataType; // stack type
    soap->fault->detail->fault = sp; // point to stack
    soap->fault->detail->__any = NULL; // no other XML data
}
return SOAP_FAULT; // return from service operation call
```

When soap_receiver_fault allocates a fault struct, this data is removed with the soap_end call (or
soap_dealloc). Note that the soap_receiver_fault function is called to allocate the fault struct and set
the fault string and detail fields, i.e. soap_receiver_fault(soap, "Stack dump", NULL). The advantage is
that this is independent of SOAP 1.1 and SOAP 1.2. However, setting the custom detail fields
requires inspecting the SOAP version used, using the soap->version attribute which is 1 for SOAP
1.1 and 2 for SOAP 1.2.

Each service operation implementation in a service application can return a SOAP Fault upon an
exception by returning an error code, see Section 7.2.1 for details and an example. In addition, a
SOAP Fault can be returned by a service application through calling the soap_send_fault function.
This is useful in case the initialization of the application fails, as illustrated in the example below:

```
int main()
{
  struct soap soap;
  soap_init(&soap);
  some initialization code
  if (initialization failed)
  {
    soap.error = soap_receiver_fault(&soap, "Init failed", NULL); // set the error condition
(SOAP_FAULT)
    soap_send_fault(&soap); // Send SOAP Fault to client
    return 0; // Terminate
  }
}
```

13   SOAP Header Processing

A predeclared standard SOAP Header data structure is generated by the gSOAP soapcpp2 tool for exchanging SOAP messages with SOAP Headers. This predeclared data structure is:

> **struct** SOAP_ENV_ _Header { };

which declares and empty header (some C and C++ compilers don't accept empty structs, use compile flag -DWITH_NOEMPTYSTRUCT to avoid these errors).

To adapt the data structure to a specific need for SOAP Header processing, a new **struct** SOAP_ENV_ _Header can be added to the header file input to the gSOAP compiler. A **class** for the SOAP Header data structure can be used instead of a **struct**.

For example, the following header can be used for transaction control:

> **struct** SOAP_ENV_ _Header
> { **char** *t_ _transaction;
> };

with client-side code:

```
    struct soap soap;
    soap_init(&soap);
...
soap.header = NULL; // do not use a SOAP Header for the request (as set with soap_init)
soap.actor = NULL; // do not use an actor (receiver is actor)
soap_call_method(&soap, ...);
if (soap.header) // a SOAP Header was received
    cout << soap.header->t_ _transaction;
// Can reset, modify, or set soap.header here before next call
soap_call_method(&soap, ...); // reuse the SOAP Header of the service response for the request
...
```

The SOAP Web service response can include a SOAP Header with a transaction number that the client is supposed to use for the next service operation invocation to the service. Therefore, the next request includes a transaction number:

```
...
<SOAP-ENV:Envelope ...>
<SOAP-ENV:Header>
<transaction xmlns="..." xsi:type="int">12345</transaction>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
...
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This is just an example and the transaction control is not a feature of SOAP but can be added on by the application layer to implement stateful transactions between clients and services. At the

client side, the `soap.actor` attribute can be set to indicate the recipient of the header (the SOAP `SOAP-ENV:actor` attribute).

A Web service can read and set the SOAP Header as follows:

```
int main()
{
  struct soap soap;
  soap.actor = NULL; // use this to accept all headers (default)
  soap.actor = "http://some/actor"; // accept headers destined for "http://some/actor" only
  soap_serve(&soap);
}
...
int method(struct soap *soap, ...)
{
  if (soap->header) // a Header was received
    ... = soap->header->t__transaction;
  else
    soap->header = soap_malloc(sizeof(struct SOAP_ENV__Header)); // alloc new header
...    soap->header->t__transaction = ...;
  return SOAP_OK;
}
```

See Section 19.2 on how to generate WSDL with the proper method-to-header-part bindings.

The `SOAP-ENV:mustUnderstand` attribute indicates the requirement that the recipient of the SOAP Header (who must correspond to the `SOAP-ENV:actor` attribute when present or when the attribute has the value `SOAP-ENV:actor="http://schemas.xmlsoap.org/soap/actor/next"`) MUST handle the Header part that carries the attribute. gSOAP handles this automatically on the background. However, an application still needs to inspect the header part's value and handle it appropriately. If a service operation in a Web service is not able to do this, it should return SOAP_MUSTUNDERSTAND to indicate this failure.

The syntax for the header file input to the gSOAP soapcpp2 compiler is extended with a special storage qualifier mustUnderstand. This qualifier can be used in the SOAP Header declaration to indicate which parts should carry a SOAP-ENV:mustUnderstand="1" attribute. For example:

```
struct SOAP_ENV__Header
{
  char *t__transaction;
  mustUnderstand char *t__authentication;
};
```

When both fields are set and soap.actor="http://some/actor" then the message contains:

```
<SOAP-ENV:Envelope ...>
<SOAP-ENV:Header>
<transaction xmlns="...">5</transaction>
<authentication xmlns="..."
  SOAP-ENV:actor="http://some/actor" SOAP-ENV:mustUnderstand="1">XX
</authentication>
```

```
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
    ...
    </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
```

14   MIME Attachments

The gSOAP toolkit supports MIME attachments as per SOAP with Attachments (SwA) specification (http://www.w3.org/TR/SOAP-attachments). In the following discussion, MIME attachment data is assumed to be resident in memory for sending operations and MIME attachments received will be stored in memory. MTOM and DIME attachments on the other hand can be streamed and therefore MTOM/DIME attachment data does not need to be stored in memory, see Section 15 and 16.

Transmitting multipart/related MIME attachments with a SOAP/XML message is accomplished with two functions, soap_set_mime and soap_set_mime_attachment. The first function is for initialization purposes and the latter function is used to specify meta data and content data for each attachment.

14.1   Sending a Collection of MIME Attachments (SwA)

The following functions should be used to set up a collection of multipart/related MIME attachments for transmission with a SOAP/XML message.

Function
void soap_set_mime(**struct** soap *soap, **const char** *boundary, **const char** *start)
This function must be called first to initialize MIME attachment send operations (receives are automatic). The function specifies a MIME boundary and start content ID used for the SOAP message body. When boundary is NULL, an appropriate MIME boundary will be choosen (important: boundaries cannot occur in the SOAP/XML message and cannot occur in any of the MIME attachments content). When a specific boundary value is provided, gSOAP will NOT verify that the boundary is valid. When start is NULL, the start ID of the SOAP message is `<SOAP-ENV:Envelope>`.
int soap_set_mime_attachment(**struct** soap *soap, **char** *ptr, size_t size, **enum** soap_mime_encoding encoding, **const char** *type, **const char** *id, **const char** *location, **const char** *description)
This function adds a new attachment to the list of attachments, where ptr and size refer to the block of memory that holds the attachment data. The encoding parameter specifies the content encoding of this block, where the value of encoding is one of SOAP_MIME_7BIT, SOAP_MIME_8BIT, SOAP_MIME_BINARY, SOAP_MIME_QUOTED_PRINTABLE, SOAP_MIME_BASE64, SOAP_MIME_IETF_TOKEN, or SOAP_MIME_X_TOKEN. These constants reflect the content encoding defined in RFC2045 and you MUST adhere to the content encoding rules defined by RFC2045. When in doubt, use SOAP_MIME_BINARY, since this encoding type covers any content. The mandatory type string parameter is the MIME type of the data. The id string parameter is the content ID of the MIME attachment. The optional location string parameter is the content location of the attachment. The optional description string parameter holds a textual description of the attachment (it may not contain any control characters). All parameter values are copied, except ptr which must point to a valid location of the attachment data during the transfer. The value SOAP_OK is returned when the attachment was added. Otherwise a gSOAP error code is returned.
void soap_clr_mime(**struct** soap *soap)
Disables MIME attachments, e.g. to avoid MIME attachments to be part of a SOAP Fault response message.

When providing a MIME boundary with soap_set_mime, you have to make sure the boundary cannot match any SOAP/XML message content. Or you can simply pass NULL and let gSOAP select a safe boundary for you.

The internal list of attachments is destroyed with soap_end, you should call this function sometime after the message exchange was completed (the content of the block of memory referred to by the ptr parameter is unaffected).

The following example shows how a multipart/related HTTP message with three MIME attachments is set up and transmitted to a server. The first attachment contains the SOAP message. The second and third attachments contain image data. In this example we let the SOAP message body refer to the attachments using `href` attributes. The **struct** claim_form data type includes a definition of a `href` attribute for this purpose.

```
struct claim_form form1, form2;
form1.href = "cid:claim061400a.tiff@claiming-it.com";
form2.href = "cid:claim061400a.jpeg@claiming-it.com";
/* initialize and enable MIME */
soap_set_mime(soap, "MIME_boundary", "<claim061400a.xml@claiming-it.com>");
/* add a base64 encoded tiff image (tiffImage points to base64 data) */
soap_set_mime_attachment(soap, tiffImage, tiffLen, SOAP_MIME_BASE64, "image/tiff",
        "<claim061400a.tiff@claiming-it.com>", NULL, NULL);
/* add a raw binary jpeg image (jpegImage points to raw data) */
```

```
soap_set_mime_attachment(soap, jpegImage, jpegLen, SOAP_MIME_BINARY, "image/jpeg",
      "<claim061400a.jpeg@claiming-it.com>", NULL, NULL);
/* send the forms as MIME attachments with this invocation */
if (soap_call_claim__insurance_claim_auto(soap, form1, form2, ...))
   // an error occurred
else
   // process response
```

Note: the above example assumes that the boundary `MIME_boundary` does not match any part of the SOAP/XML message.

The claim__form struct is declared in the gSOAP header file as:

```
struct claim__form
{ @char *href;
};
```

This data type defines the parameter data of the operation. The claim forms in the SOAP/XML message consist of `hrefs` to the claim forms attached. The produced message is similar to the last example shown in the SOAP with Attachments specification (http://www.w3.org/TR/SOAP-attachments). Note that the use of `href` or other attributes for referring to the MIME attachments is optional according to the SwA standard.

To associate MIME attachments with the request and response of a service operation in the generated WSDL, please see Section 16.1.

The server-side code to transmit MIME attachments back to a client is similar:

```
int claim__insurance_claim_auto(struct soap *soap, ...)
{
   soap_set_mime(soap, NULL, NULL); // enable MIME
   // add a HTML document (htmlDoc points to data, where the HTML doc is stored in compliance
with 7bit encoding RFC2045)
   if (soap_set_mime_attachment(soap, htmlDoc, strlen(htmlDoc), SOAP_MIME_7BIT, "text/html",
      "<claim061400a.html@claiming-it.com>", NULL, NULL))
   {
      soap_clr_mime(soap); // don't want fault with attachments
      return soap->error;
   }
   return SOAP_OK;
}
```

It is also possible to attach data to a SOAP fault message.

Caution: DIME in MIME is supported. However, gSOAP will not verify whether the MIME boundary is present in the DIME attachments and therefore will not select a boundary that is guaranteed to be unique. Therefore, you must provide a MIME boundary with soap_set_mime that is unique when using DIME in MIME.

14.2 Retrieving a Collection of MIME Attachments (SwA)

MIME attachments are automatically parsed and stored in memory. After receiving a set of MIME attachments, either at the client-side or the server-side, the list of MIME attachments can be traversed to extract meta data and the attachment content. The first attachment in the collection of MIME attachments always contains meta data about the SOAP message itself (because the SOAP message was processed the attachment does not contain any useful data).

To traverse the list of MIME attachments in C, you use a loop similar to:

```
struct soap_multipart *attachment;
for (attachment = soap.mime.list; attachment; attachment = attachment->next)
{
  printf("MIME attachment:\n");
  printf("Memory=%p\n", (*attachment).ptr);
  printf("Size=%ul\n", (*attachment).size);
  printf("Encoding=%d\n", (int)(*attachment).encoding);
  printf("Type=%s\n", (*attachment).type?(*attachment).type:"null");
  printf("ID=%s\n", (*attachment).id?(*attachment).id:"null");
  printf("Location=%s\n", (*attachment).location?(*attachment).location:"null");
  printf("Description=%s\n", (*attachment).description?(*attachment).description:"null");
}
```

C++ programmers can use an iterator instead, as in:

```
for (soap_multipart::iterator attachment = soap.mime.begin(); attachment != soap.mime.end();
++attachment)
{
  cout << "MIME attachment:" << endl;
  cout << "Memory=" << (void*)(*attachment).ptr << endl;
  cout << "Size=" << (*attachment).size << endl;
  cout << "Encoding=" << (*attachment).encoding << endl;
  cout << "Type=" << ((*attachment).type?(*attachment).type:"null") << endl;
  cout << "ID=" << ((*attachment).id?(*attachment).id:"null") << endl;
  cout << "Location=" << ((*attachment).location?(*attachment).location:"null") << endl;
  cout << "Description=" << ((*attachment).description?(*attachment).description:"null") << endl;
}
```

Note: keep in mind that the first attachment is associated with the SOAP message and you may want to ignore it.

A call to soap_end removes all of the received MIME data. To preserve an attachment in memory, use soap_unlink on the ptr field of the soap_multipart struct. Recall that the soap_unlink function is commonly used to prevent deallocation of deserialized data.

15 DIME Attachments

The gSOAP toolkit supports DIME attachments as per DIME specification, see http://msdn.microsoft.com/library/us/dnglobspec/html/draft-nielsen-dime-02.txt

Applications developed with gSOAP can transmit binary DIME attachments with or without streaming messages. Without streaming, all data is stored and retrieved in memory, which can be prohibitive when transmitting large files on small devices. The maximum DIME attachment size is limited to 8 MB by default as set with SOAP_MAXDIMESIZE in stdsoap2.h. In contrast, with DIME streaming, data handlers are used to pass the data to and from a resource, such as a file or device. With DIME output streaming, raw binary data is send from a data source in chunks on the fly without buffering the entire content to save memory. With DIME input streaming, raw binary data will be passed to data handlers (callbacks).

15.1   Sending a Collection of DIME Attachments

The following functions can be used to explicitly set up a collection of DIME attachments for transmission with a SOAP/XML message body. The attachments can be streamed, as described in Section 15.4. Without streaming, each attachment must refer to a block of data in memory.

Function
void soap_set_dime(**struct** soap *soap)
This function must be called first to initialize DIME attachment send operations (receives are automatic).
int soap_set_dime_attachment(**struct** soap *soap, **char** *ptr, size_t size, **const char** *type, **const char** *id, **unsigned short** optype, **const char** *option)
This function adds a new attachment to the list of attachments, where ptr and size refer to the block of memory that holds the attachment data (except when DIME streaming callback handlers are used as described in Section 15.4. The type string parameter is the MIME type of the data. The id string parameter is the content ID of the DIME attachment. The option string parameter holds optional text (gSOAP supports DIME options, but it can send only one) and optype is a user-defined option type (as per DIME option specification format). All parameter values are copied, except ptr. The value SOAP_OK is returned when the attachment was added. Otherwise a gSOAP error code is returned.
void soap_clr_dime(**struct** soap *soap)
Disables DIME attachments, unless the serialized SOAP message contains attachments for transmission.

These functions allow DIME attachments to be added without requiring message body references. This is also referred to as the open DIME attachment style. The closed attachment style requires all DIME attachments to be referenced from the SOAP message body with `href` (or similar) references. For the closed style, gSOAP supports an automatic binary data serialization method, see Section 15.3.

15.2   Retrieving a Collection of DIME Attachments

DIME attachments are automatically parsed and stored in memory (or passed to the streaming handlers, when applicable). After receiving a set of DIME attachments, either at the client-side or the server-side, the list of DIME attachments can be traversed to extract meta data and the attachment content.

To traverse the list of DIME attachments in C, you use a loop similar to:

```
struct soap_multipart *attachment;
for (attachment = soap.dime.list; attachment; attachment = attachment->next)
{
  printf("DIME attachment:\n");
  printf("Memory=%p\n", (*attachment).ptr);
  printf("Size=%ul\n", (*attachment).size);
  printf("Type=%s\n", (*attachment).type?(*attachment).type:"null");
  printf("ID=%s\n", (*attachment).id?(*attachment).id:"null");
}
```

C++ programmers can use an iterator instead, as in:

```
for (soap_multipart::iterator attachment = soap.dime.begin(); attachment != soap.dime.end(); ++at-
tachment)
{
  cout << "DIME attachment:" << endl;
  cout << "Memory=" << (void*)(*attachment).ptr << endl;
  cout << "Size=" << (*attachment).size << endl;
  cout << "Type=" << ((*attachment).type?(*attachment).type:"null") << endl;
  cout << "ID=" << ((*attachment).id?(*attachment).id:"null") << endl;
}
```

The options field is available as well. The options content is formatted according to the DIME specification: the first two bytes are reserved for the option type, the next two bytes store the size of the option data, followed by the (binary) option data.

A call to soap_end removes all of the received DIME data. To preserve an attachment in memory, use soap_unlink on the ptr field of the soap_multipart struct. Recall that the soap_unlink function is commonly used to prevent deallocation of deserialized data.

15.3   Serializing Binary Data in DIME

Binary data stored in extended xsd:base64Binary and xsd:hexBinary types can be serialized and deserialized as DIME attachments. These attachments will be transmitted prior to the sequence of secondary DIME attachments defined by the user with soap_set_dime_attachment as explained in the previous section. The serialization process is automated and DIME attachments will be send even when soap_set_dime or soap_set_dime_attachment are not used.

The xsd:base64Binary XSD type is defined in gSOAP as a struct or class by

```
struct xsd__base64Binary
{
  unsigned char *__ptr; // pointer to raw binary data
  int __size; // size of the block of data
};
```

To enable serialization of the data in DIME, we extend this type with three additional fields:

```
struct xsd__base64Binary
{
```

```
    unsigned char *__ptr;
    int __size;
    char *id;
    char *type;
    char *options;
};
```

The three additional fields consist of an id field for attachment referencing (typically a content id (CID) or UUID), a type field to specify the MIME type of the binary data, and an options field to piggy-back additional information with a DIME attachment. The order of the declaration of the fields is significant. In addition, no other fields or methods may be declared before any of these fields in the struct/class, but additional fields and methods may appear after the field declarations. An extended xsd__hexBinary declaration is similar.

The id and type fields contain text. The set the DIME-specific options field, you can use the soap_dime_option function:

char *soap_dime_option(**struct** soap *soap, **unsigned short** type, **const char** *option)

returns a string with this encoding. For example

```
struct xsd__base64Binary image;
image.__ptr = ...;
image.__size = ...;
image.id = "uuid:09233523-345b-4351-b623-5dsf35sgs5d6";
image.type = "image/jpeg";
image.options = soap_dime_option(soap, 0, "My wedding picture");
```

When either the id or type field values are non-NULL at run time, the data will be serialized as a DIME attachment. The SOAP/XML message refers to the attachments using href attributes. This generally works will with SOAP RPC, because href attributes are permitted. However, with document/literal style the referencing mechanism must be explicitly defined in the schema of the binary type. The gSOAP declaration of an extended binary type is

```
struct ns__myBinaryDataType
{
    unsigned char *__ptr;
    int __size;
    char *id;
    char *type;
    char *options;
};
```

C++ programmers can use inheritance instead of textual extension required in C, as in

```
class xsd__base64Binary
{
    unsigned char *__ptr;
    int __size;
```

```
  };
  class ns__myBinaryDataType : xsd__base64Binary
  {
    char *id;
    char *type;
    char *options;
  };
```

This defines an extension of `xsd:base64Binary`, such that the data can be serialized as DIME attachments using `href` attributes for referencing. When a different attribute name is in fact used, it must be explicitly defined:

```
  //gsoap WSref schema import: http://schemas.xmlsoap.org/ws/2002/04/reference/
  struct ns__myBinaryDataType
  {
    unsigned char *__ptr;
    int __size;
    char *id;
    char *type;
    char *options;
    @char *WSref__location;
  };
```

The example above uses the `location` attribute defined in the content reference schema, as defined in one of the vendor's specific WSDL extensions for DIME (http://www.gotdotnet.com/team/xml_wsspecs/dime/WSI Extension-for-DIME.htm).

When receiving DIME attachments, the DIME meta data and binary data content is stored in binary data types only when the XML parts of the message uses `href` attributes to refer to these attachments. The gSOAP toolkit may support automatic (de)serialization with other user-defined (or WSDL-defined) attributes in future releases.

Messages may contain binary data that references external resources not provided as attachments. In that case, the `__ptr` field is NULL and the `id` field refers to the external data source.

The `dime_id_format` attribute of the current gSOAP run-time context can be set to the default format of DIME id fields. The format string MUST contain a `%d` format specifier (or any other **int**-based format specifier). The value of this specifier is a non-negative integer, with zero being the value of the DIME attachment id for the SOAP message. For example,

```
  struct soap soap;
  soap_init(&soap);
  soap.dime_id_format = "uuid:09233523-345b-4351-b623-5dsf35sgs5d6-%x";
```

As a result, all attachments with a NULL `id` field will use a gSOAP-generated id value based on the format string.

Caution: Care must be taken not to introduce duplicate content id values, when assigning content id values to the id fields of DIME extended binary data types. Content ids must be unique.

15.4   Streaming DIME

Streaming DIME is achieved with callback functions to fetch and store data during transmission. Three function callbacks for streaming DIME output and three callbacks for streaming DIME input are available.

Callback (function pointer)

void *(*soap.fdimereadopen)(**struct** soap *soap, **void** *handle, **const char** *id, **const char** *type, **const char** *options)

Called by the gSOAP run-time DIME attachment sender to start reading from a (binary) data source for outbound transmission. The content will be read from the application's data source in chunks using the fdimeread callback and streamed into the SOAP/XML/DIME output stream. The handle contains the value of the __ptr field of an attachment struct/class, which could be a pointer to specific information such as a file descriptor or a pointer to a string to be passed to this callback. Both __ptr and __size fields should have been set by the application prior to the serialization of the content. The id, type, and options arguments are the DIME id, type, and options, respectively. The callback should return handle, or another pointer value which will be passed as a handle to fdimeread and fdimereadclose. The callback should return NULL and set soap->error when an error occurred. The callback should return NULL (and not set soap->error) when this particular DIME attachment is not to be streamed.

size_t (*soap.fdimeread)(**struct** soap *soap, **void** *handle, **char** *buf, size_t len)

Called by the gSOAP run-time DIME attachment sender to read more data from a (binary) data source for streaming into the output stream. The handle contains the value returned by the fdimeread-open callback. The buf argument is the buffer of length len into which a chunk of data should be stored. The actual amount of data stored in the buffer may be less than len and this amount should be returned by the application. A return value of 0 indicates an error (the callback may set soap->errnum to errno). The __size field of the attachment struct/class should have been set by the application prior to the serialization of the content. The value of __size indicates the total size of the content to be transmitted. When the __size is zero then DIME chunked transfers can be used under certain circumstances to stream content without prior determination of attachment size, see Section 15.5 below.

void(*soap.fdimereadclose)(**struct** soap *soap, **void** *handle)

Called by the gSOAP run-time DIME attachment sender at the end of the streaming process to close the data source. The handle contains the value returned by the fdimereadopen callback. The fdimewriteclose callback is called after successfully transmitting the data or when an error occurred.

void *(*soap.fdimewriteopen)(**struct** soap *soap, **const char** *id, **const char** *type, **const char** *options)

Called by the gSOAP run-time DIME attachment receiver to start writing an inbound DIME attachment to an application's data store. The content is streamed into an application data store through multiple fdimewrite calls from the gSOAP attachment receiver. The id, type, and options arguments are the DIME id, type, and options respectively. The callback should return a handle which is passed to the fdimewrite and fdimewriteclose callbacks. The __ptr field of the attachment struct/class is set to the value of this handle. The __size field is set to the total size of the attachment after receiving the entire content. The size is unknown in advance because DIME attachments may be chunked.

int (*soap.fdimewrite)(**struct** soap *soap, **void** *handle, **const char** *buf, size_t len)

Called by the gSOAP run-time DIME attachment receiver to write part of an inbound DIME attachment to an application's data store. The handle contains the value returned by the fdimewriteopen callback. The buf argument contains the data of length len. The callback should return a gSOAP error code (e.g. SOAP_OK when no error occurred).

void(*soap.fdimewriteclose)(**struct** soap *soap, **void** *handle)

Called by the gSOAP run-time DIME attachment receiver at the end of the streaming process to close the data store. The fdimewriteclose callback is called after successfully receiving the data or when an error occurred. The handle contains the value returned by the fdimewriteopen callback.

In addition, a **void***user field in the **struct** soap data structure is available to pass user-defined data to the callbacks. This way, you can set soap.user to point to application data that the callbacks

need such as a file name for example.

The following example illustrates the client-side initialization of an image attachment struct to stream a file into a DIME attachment:

```
int main()
{
  struct soap soap;
  struct xsd__base64Binary image;
  FILE *fd;
  struct stat sb;
  soap_init(&soap);
  if (!fstat(fileno(fd), &sb) && sb.st_size > 0)
  { // because we can get the length of the file, we can stream it
    soap.fdimereadopen = dime_read_open;
    soap.fdimereadclose = dime_read_close;
    soap.fdimeread = dime_read;
    image.__ptr = (unsigned char*)fd; // must set to non-NULL (this is our fd handle which we
need in the callbacks)
    image.__size = sb.st_size; // must set size
  }
  else
  { // don't know the size, so buffer it
    size_t i;
    int c;
    image.__ptr = (unsigned char*)soap_malloc(&soap, MAX_FILE_SIZE);
    for (i = 0; i < MAX_FILE_SIZE; i++)
    {
      if ((c = fgetc(fd)) == EOF)
        break;
      image.__ptr[i] = c;
    }
    fclose(fd);
    image.__size = i;
  }
  image.type = "image/jpeg";
  image.options = soap_dime_option(&soap, 0, "My picture");
  soap_call_ns__method(&soap, ...);
  ...
}
void *dime_read_open(struct soap *soap, void *handle, const char *id, const char *type, const
char *options)
{ return handle;
}
void dime_read_close(struct soap *soap, void *handle)
{ fclose((FILE*)handle);
}
size_t dime_read(struct soap *soap, void *handle, char *buf, size_t len)
{ return fread(buf, 1, len, (FILE*)handle);
}
```

The following example illustrates the streaming of a DIME attachment into a file by a client:

```c
int main()
{ struct soap soap;
  soap_init(&soap);
  soap.fdimewriteopen = dime_write_open;
  soap.fdimewriteclose = dime_write_close;
  soap.fdimewrite = dime_write;
  soap_call_ns_ _method(&soap, ...);
  ...
}
void *dime_write_open(struct soap *soap, const char *id, const char *type, const char *options)
{
  FILE *handle = fopen("somefile", "wb");
  if (!handle)
  {
    soap->error = SOAP_EOF;
    soap->errnum = errno; // get reason
  }
  return (void*)handle;
}
void dime_write_close(struct soap *soap, void *handle)
{ fclose((FILE*)handle);
}
int dime_write(struct soap *soap, void *handle, const char *buf, size_t len)
{
  size_t nwritten;
  while (len)
  {
    nwritten = fwrite(buf, 1, len, (FILE*)handle);
    if (!nwritten)
    {
      soap->errnum = errno; // get reason
      return SOAP_EOF;
    }
    len -= nwritten;
    buf += nwritten;
  }
  return SOAP_OK;
}
```

Note that compression can be used with DIME to compress the entire message. However, compression requires buffering to determine the HTTP content length header, which cancels the benefits of streaming DIME. To avoid this, you should use chunked HTTP (with the output-mode SOAP_IO_CHUNK flag) with compression and streaming DIME. At the server side, when you set SOAP_IO_CHUNK before calling soap_serve, gSOAP will automatically revert to buffering (SOAP_IO_STORE flag is set). You can check this flag with (soap->omode & SOAP_IO) == SOAP_IO_CHUNK to see if the client accepts chunking. More information about streaming chunked DIME can be found in Section 15.5.

Caution: The options field is a DIME-specific data structure, consisting of a 4 byte header containing the option type info (hi byte, lo byte), option string length (hi byte, lo byte), followed by a non-'\0' terminated string. The gSOAP DIME handler recognizes one option at most.

15.5 Streaming Chunked DIME

gSOAP automatically handles inbound chunked DIME attachments (streaming or non-streaming). To transmit outbound DIME attachments, the attachment sizes MUST be determined in advance to calculate HTTP message length required to stream DIME over HTTP. However, gSOAP also supports the transmission of outbound chunked DIME attachments without prior determination of DIME attachment sizes when certain conditions are met. These conditions require either non-HTTP transport (use the output-mode SOAP_ENC_XML flag), or chunked HTTP transport (use the output-mode SOAP_IO_CHUNK flag). You can also use the SOAP_IO_STORE flag (which is also used automatically with compression to determine the HTTP content length header) but that cancels the benefits of streaming DIME.

To stream chunked DIME, set the __size field of an attachment to zero and enable HTTP chunking. The DIME fdimeread callback then fetches data in chunks and it is important to fill the entire buffer unless the end of the data has been reached and the last chunk is to be send. That is, fdimeread should return the value of the last len parameter and fill the entire buffer buf for all chunks except the last.

15.6 WSDL Bindings for DIME Attachments

The wsdl2h WSDL parser recognizes DIME attachments and produces an annotated header file. Both open and closed layouts are supported for transmitting DIME attachments. For closed formats, all DIME attachments must be referenced from the SOAP message, e.g. using hrefs with SOAP encoding and using the application-specific reference attribute included in the base64Binary struct/class for doc/lit.

The gSOAP compiler soapcpp2 does not produce a WSDL with DIME extensions. DIME is an older binary format that has no WSDL protocol support, unlike MIME and MTOM.

16 MTOM Attachments

MTOM (Message Transmission Optimization Mechanism) is a relatively new format for transmitting attachments with SOAP messages (see `http://www.w3.org/TR/soap12-mtom`). MTOM is a W3C working draft as of this writing. MTOM attachments are essentially MIME attachments with standardized mechanisms for cross referencing attachments from the SOAP body, which is absent in (plain) MIME attachments and optional with DIME attachments.

Unlike the name suggests, the speed by which attached data is transmitted is not increased compared to MIME, DIME, or even XML encoded base64 data (at least the performance differences in gSOAP will be small). The advantage of the format is the standardized attachment reference mechanism, which should improve interoperability.

The MTOM specification mandates SOAP 1.2 and the use of the XOP namespace. The XOP Include element `xop:Include` is used to reference attachment(s) from the SOAP message body.

Because references from within the SOAP message body to attachments are mandatory with MTOM, the implementation of the serialization and deserialization of MTOM MIME attachments

in gSOAP uses the extended binary type comparable to DIME support in gSOAP. This binary type is predefined in the import/xop.h file:

```
//gsoap xop schema import: http://www.w3.org/2004/08/xop/include
struct _xop__Include
{
    unsigned char *__ptr;
    int __size;
    char *id;
    char *type;
    char *options;
};
typedef struct _xop__Include _xop__Include;
```

The additional id, type, and option fields enable MTOM attachments for the data pointed to by __ptr of size __size. The process for sending and receiving MTOM XOP attachments is fully automated. The id field references the attachment (typically a content id CID or UUID). When set to NULL, gSOAP assigns a unique CID. The type field specifies the required MIME type of the binary data, and the optional options field can be used to piggy-back descriptive text with an attachment. The order of the declaration of the fields is significant.

You can explicitly import the xop.h in your header file to use the MTOM attachments in your service, for example:

```
#import "import/soap12.h"
/* alternatively, without the import above, use:
//gsoap SOAP-ENV schema namespace: http://www.w3.org/2003/05/soap-envelope
//gsoap SOAP-ENC schema namespace: http://www.w3.org/2003/05/soap-encoding
/
#import "import/xop.h"
#import "import/xmime5.h"

//gsoap x schema namespace: http://my.first.mtom.net
struct x__myData
{
    _xop__Include xop__Include; // attachment
    @char *xmime5__contentType; // and its contentType
};
int x__myMTOMtest(struct x__myData *in, struct x__myData *out);
```

As you can see, there is really no difference between the specification of MTOM and DIME attachments in a gSOAP header file. Except that you MUST use SOAP 1.2 and the xop__Include element.

When an instance of x__myDataType is serialized and either or both the id and type fields are non-NULL, the data is transmitted as MTOM MIME attachment if the SOAP_ENC_MTOM flag is set in the gSOAP's soap struct context:

```
struct soap *soap = soap_new1(SOAP_ENC_MTOM);
```

Without this flag, the attachments will be transmitted in DIME format (Section 15). If your current clients and services are based on non-streaming DIME attachments using the SOAP body reference mechanism (thus, without using the soap_set_dime_attachment function) or plain base64 binary XML data elements, it is very easy to adopt MTOM by renaming the binary types to xop_Include and using the SOAP_ENC_MTOM flag with the SOAP 1.2 namespace.

16.1 Generating MultipartRelated MIME Attachment Bindings in WSDL

To generate multipartRelated bindings in the WSDL file, use the //gsoap ... service method-mime-type directive (see also Section 19.2. The directive can be repeated for each attachment you want to associate with a method's request and response messages.

For example:

```
#import "import/soap12.h"
#import "import/xop.h"
#import "import/xmime5.h"

//gsoap x schema namespace: http://my.first.mtom.net
struct x__myData
{
   _xop__Include xop__Include; // attachment
   @char *xmime5__contentType; // and its contentType
};
//gsoap x service method-mime-type: myMTOMtest text/xml
int x__myMTOMtest(struct x__myData *in, struct x__myData *out);
```

The //gsoap x service method-mime-type directive indicates that this operation accepts `text/xml` MIME attachments. See the SOAP-with-Attachment specification for the MIME types to use (for example, `*/*` is a wildcard). If the operation has more than one attachment, just repeat this directive for each attachment you want to bind to the operation.

To bind attachments only to the request message of an operation, use //gsoap x service method-input-mime-type. Similarly, to bind attachments only to the response message of an operation, use //gsoap x service method-ouput-mime-type.

The wsdl2h WSDL parser recognizes MIME attachments and produces an annotated header file. However, the ordering of MIME parts in the multipartRelated elements is not reflected in the header file. Application developers should adhere the standards and ensure that multipart/related attachments are transmitted in compliance with the WSDL operation declarations.

16.2 Sending and Receiving MTOM Attachments

A receiver must be informed to recognize MTOM attachments by setting the SOAP_ENC_MTOM flag of the gSOAP context. Otherwise, the regular MIME attachment mechanism (SwA) will be used to store attachments.

When using wsdl2h to build clients and/or services, you should use the typemap.dat file included in the distribution package. The typemap.dat file defines the XOP namespace and XML MIME

namespaces as imported namespaces:

```
xop = <http://www.w3.org/2004/08/xop/include>
xmime5 = <http://www.w3.org/2005/05/xmlmime>
xmime4 = <http://www.w3.org/2004/11/xmlmime>
```

The wsdl2h tool uses the typemap.dat file (see also option -t) to convert WSDL into a gSOAP header file. In this case we don't want the wsdl2h tool to read the XOP schema and translate it, since we have a pre-defined _xop__Include element to handle XOP for MTOM. This _xop__Include element is defined in xop.h. Therefore, the bindings shown above will not translate the XOP and XML MIME schemas to code, but generates #import statements instead:

```
#import "xop.h"
#import "xmime5.h"
```

The #import statements are only added for those namespaces that are actually used by the service.

Let's take a look at an example. The wsdl2h importer generates a header file with #import "xop.h" from a WSDL that references XOP, for example:

```
#import "xop.h"
#import "xmime5.h"
struct ns__Data
{
  _xop__Include xop__Include;
  @char *xmime5__contentType;
};
```

Suppose the WSDL defines an operation:

```
int ns__echoData(struct ns__Data *in, struct ns__Data *out);
```

After generating the stubs/proxies with the soapcpp2 compiler, we can invoke the stub at the client side with:

```
struct soap *soap = soap_new1(SOAP_ENC_MTOM);
struct ns__Data data;
data.xop__Include.__ptr = (unsigned char*)"<b>Hello world!</b>";
data.xop__Include.__size = 20;
data.xop__Include.id = NULL; // CID automatically generated by gSOAP engine
data.xop__Include.type = "text/html"; // MIME type
data.xop__Include.options = NULL; // no descriptive info added
data.xmime5__contentType = "text/html"; // MIME type
if (soap_call_ns__echoData(soap, endpoint, action, &data, &data))    soap_print_fault(soap, stderr);
else
  printf("Got data\n");
soap_destroy(soap); // remove deserialized class instances
soap_end(soap); // remove temporary and deserialized data
soap_free(soap); // detach and free context
```

Note that the xop__Include.type field must be set to transmit MTOM attachments, otherwise plain base64 XML will be used.

At the server side, we show an example of an operation handler that just copies the input data to output:

```
int ns__echoData(struct soap *soap, struct ns__Data *in, struct ns__data *out)
{
    *out = *in;
    return SOAP_OK;
}
```

The server must use the SOAP_ENC_MTOM flag to initialize the soap struct to receive and send MTOM attachments.

16.3   Streaming MTOM/MIME

Streaming MTOM/MIME is achieved with callback functions to fetch and store data during transmission. Three function callbacks for streaming MTOM/MIME output and three callbacks for streaming MTOM/MIME input are available.

Callback (function pointer)

void *(*soap.fmimereadopen)(**struct** soap *soap, **void** *handle, **const char** *id, **const char** *type, **const char** *description)

Called by the gSOAP run-time MTOM/MIME attachment sender to start reading from a (binary) data source for outbound transmission. The content will be read from the application's data source in chunks using the fmimeread callback and streamed into the SOAP/XML/MTOM/MIME output stream. The handle contains the value of the __ptr field of an attachment struct/class, which could be a pointer to specific information such as a file descriptor or a pointer to a string to be passed to this callback. Both __ptr and __size fields should have been set by the application prior to the serialization of the content. The id, type, and description arguments are the MTOM/MIME id, type, and description, respectively. The callback should return handle, or another pointer value which will be passed as a handle to fmimeread and fmimereadclose. The callback should return NULL and set soap->error when an error occurred. The callback should return NULL (and not set soap->error) when this particular MTOM/MIME attachment is not to be streamed.

size_t (*soap.fmimeread)(**struct** soap *soap, **void** *handle, **char** *buf, size_t len)

Called by the gSOAP run-time MTOM/MIME attachment sender to read more data from a (binary) data source for streaming into the output stream. The handle contains the value returned by the fmimereadopen callback. The buf argument is the buffer of length len into which a chunk of data should be stored. The actual amount of data stored in the buffer may be less than len and this amount should be returned by the application. A return value of 0 indicates an error (the callback may set soap->errnum to errno). The __size field of the attachment struct/class should have been set by the application prior to the serialization of the content. The value of __size indicates the total size of the content to be transmitted. When the __size is zero then MTOM/MIME chunked transfers can be used under certain circumstances to stream content without prior determination of attachment size, see Section 16.5 below.

void(*soap.fmimereadclose)(**struct** soap *soap, **void** *handle)

Called by the gSOAP run-time MTOM/MIME attachment sender at the end of the streaming process to close the data source. The handle contains the value returned by the fmimereadopen callback. The fmimewriteclose callback is called after successfully transmitting the data or when an error occurred.

void *(*soap.fmimewriteopen)(**struct** soap *soap, **void** *handle, **const char** *id, **const char** *type, **const char** *description, **enum** soap_mime_encoding encoding)

Called by the gSOAP run-time MTOM/MIME attachment receiver to start writing an inbound MTOM/MIME attachment to an application's data store. The content is streamed into an application data store through multiple fmimewrite calls from the gSOAP attachment receiver. The handle argument is normally NULL, unless soap_get_mime_attachment is used that passes the handle to the callback, see Section 16.4. The id, type, and description arguments are the MTOM/MIME id, type, and description respectively. The encoding enumeration value indicates the MIME content transfer encoding, which is one of SOAP_MIME_NONE, SOAP_MIME_7BIT, SOAP_MIME_8BIT, SOAP_MIME_BINARY, SOAP_MIME_QUOTED_PRINTABLE, SOAP_MIME_BASE64, SOAP_MIME_IETF_TOKEN, SOAP_MIME_X_TOKEN. Content decoding may have to be considered by the application based on this value. The callback should return a non-NULL handle which is passed to the fmimewrite and fmimewriteclose callbacks. The __ptr field of the attachment struct/class is set to the value of this handle. The __size field is set to the total size of the attachment after receiving the entire content. The size is unknown in advance because MTOM/MIME attachments may be chunked.

int (*soap.fmimewrite)(**struct** soap *soap, **void** *handle, **const char** *buf, size_t len)

Called by the gSOAP run-time MTOM/MIME attachment receiver to write part of an inbound MTOM/MIME attachment to an application's data store. The handle contains the value returned by the fmimewriteopen callback. The buf argument contains the data of length len. The callback should return a gSOAP error code (e.g. SOAP_OK when no error occurred).

void(*soap.fmimewriteclose)(**struct** soap *soap, **void** *handle)

Called by the gSOAP run-time MTOM/MIME attachment receiver at the end of the streaming process to close the data store. The fmimewriteclose callback is called after successfully receiving the data or when an error occurred. The handle contains the value returned by the fmimewriteopen callback.

In addition, a **void***user field in the **struct** soap data structure is available to pass user-defined data to the callbacks. This way, you can set soap.user to point to application data that the callbacks need such as a file name for example.

The following example illustrates the client-side initialization of an image attachment struct to stream a file into a MTOM attachment without HTTP chunking (HTTP streaming chunked MTOM transfer is presented in Section 16.5):

```
int main()
{
   struct soap soap;
   struct xsd__base64Binary image;
   FILE *fd;
   struct stat sb;
   soap_init1(&soap, SOAP_ENC_MTOM); // mandatory to enable MTOM
   if (!fstat(fileno(fd), &sb) && sb.st_size > 0)
   { // because we can get the length of the file, we can stream it without chunking
      soap.fmimereadopen = mime_read_open;
      soap.fmimereadclose = mime_read_close;
      soap.fmimeread = mime_read;
      image.__ptr = (unsigned char*)fd; // must set to non-NULL (this is our fd handle which we
need in the callbacks)
      image.__size = sb.st_size; // must set size
   }
   else
   { // don't know the size, so buffer it
      size_t i;
      int c;
      image.__ptr = (unsigned char*)soap_malloc(&soap, MAX_FILE_SIZE);
      for (i = 0; i < MAX_FILE_SIZE; i++)
      {
         if ((c = fgetc(fd)) == EOF)
            break;
         image.__ptr[i] = c;
      }
      fclose(fd);
      image.__size = i;
   }
   image.type = "image/jpeg"; // MIME type
   image.options = "This is my picture"; // description of object
   soap_call_ns__method(&soap, ...);
   ...
}
void *mime_read_open(struct soap *soap, void *handle, const char *id, const char *type, const
char *description)
{ return handle;
}
void mime_read_close(struct soap *soap, void *handle)
{ fclose((FILE*)handle);
}
size_t mime_read(struct soap *soap, void *handle, char *buf, size_t len)
{ return fread(buf, 1, len, (FILE*)handle);
```

```
    }
```

The following example illustrates the streaming of a MTOM/MIME attachment into a file by a client:

```
    int main()
    { struct soap soap;
      soap_init(&soap);
      soap.fmimewriteopen = mime_write_open;
      soap.fmimewriteclose = mime_write_close;
      soap.fmimewrite = mime_write;
      soap_call_ns__method(&soap, ...);
      ...
    }
    void *mime_write_open(struct soap *soap, const char *id, const char *type, const char *descrip-
    tion, enum soap_mime_encoding encoding)
    {
      FILE *handle = fopen("somefile", "wb");
      // We ignore the MIME content transfer encoding here, but should check
      if (!handle)
      {
        soap->error = SOAP_EOF;
        soap->errnum = errno; // get reason
      }
      return (void*)handle;
    }
    void mime_write_close(struct soap *soap, void *handle)
    { fclose((FILE*)handle);
    }
    int mime_write(struct soap *soap, void *handle, const char *buf, size_t len)
    {
      size_t nwritten;
      while (len)
      {
        nwritten = fwrite(buf, 1, len, (FILE*)handle);
        if (!nwritten)
        {
          soap->errnum = errno; // get reason
          return SOAP_EOF;
        }
        len -= nwritten;
        buf += nwritten;
      }
      return SOAP_OK;
    }
```

Note that compression can be used with MTOM/MIME to compress the entire message. However, compression requires buffering to determine the HTTP content length header, which cancels the benefits of streaming MTOM/MIME. To avoid this, you should use chunked HTTP (with the output-mode SOAP_IO_CHUNK flag) with compression and streaming MTOM/MIME. At the server side, when you set SOAP_IO_CHUNK before calling soap_serve, gSOAP will automatically revert to

buffering (SOAP_IO_STORE flag is set). You can check this flag with (soap->omode & SOAP_IO) == SOAP_IO_CHUNK to see if the client accepts chunking. More information about streaming chunked MTOM/MIME can be found in Section 16.5.

Note that the example above for mime_read uses a handle that points to the open file FILE*. The simple example above is not recommended when the platform imposes a limit on the number of open file descriptors. You can use the handle to pass along more information than just the file descriptor. So for example, when the number of open file descriptors is limited on your platform, you should let the handle point to a structure with file-related information. The C++ example below illustrates this:

```
file.xop__Include = soap_new__xop__Include(soap);
file.xop__Include->id = NULL;
file.xop__Include->type = type;
file.xop__Include->options = NULL;

file.xmime5__contentType = type;
file.filename = filename;

// The object holding all information to read data
FileStreamIn *ins = new FileStreamIn(errorhandler);
ins->setFilePath(path);
ins->setFileName(filename);

file.xop__Include->__size = size;
file.xop__Include->__ptr = (unsigned char*)ins;
```

To read the MTOM data for transmission:

```
void *mime_read_open(struct soap *soap, void *handle, const char *id, const char *type, const char *description)
{
  if (!handle)
    return NULL;
  FileStreamIn *ins = (FileStreamIn*)handle;
  if (!ins->open())
  {
    soap->error = SOAP_ERR;
    return NULL;
  }
  return handle;
}
void mime_read_close(struct soap *soap, void *handle)
{
  if (!handle)
    return;
  FileStreamIn *ins = (FileStreamIn*)handle;
  delete ins;
}
size_t mime_read(struct soap *soap, void *handle, char *buf, size_t len)
{
```

```
    if (!handle)
        return 0;
    FileStreamIn *ins = (FileStreamIn*)handle;
    size_t nread = ins->read(buf, len);
    if (ins->streamError())
    {
        soap->error = ins->streamError();
        return 0;
    }
    return nread;
}
```

16.4 Redirecting Inbound MTOM/MIME Streams Based on SOAP Body Content

When it is preferable or required to redirect inbound MTOM/MIME attachment streams based on SOAP message body content, where for example the names of the resources are listed in the SOAP message body, an alternative mechanism must be used. This mechanism can be used both at the client and server side.

Because the routing of the streams is accomplished with explicit function calls, this method should only be used when required and should not be considered optional. That is, when you enable this method, you MUST check for pending MTOM/MIME attachments and handle them appropriately. This is true even when you don't expect MTOM/MIME attachments in the payload, because the peer may trick you by sending attachments anyway and you should be prepared to accept or reject them.

The explicit MTOM/MIME streaming mechanism consists of three API functions:

Function
void soap_post_check_mime_attachments(**struct** soap *soap)
Enables post-message body inbound streaming MTOM/MIME attachments. The presence of attachments must be explicitly checked using the function below.
int soap_check_mime_attachments(**struct** soap *soap)
Should be called after a client-side call (e.g. soap_call_ns_ _method) to check the presence of attachments. Returns 1 (true) when attachments are present. If present, each attachment MUST be processed with the function below.
struct soap_multipart *soap_get_mime_attachment(**struct** soap *soap, **void** *handle)
Parses an attachment and invokes the MIME callbacks (when set). The handle parameter is passed to fmimewriteopen. The handle may contain any data that is extracted from the SOAP message body to guide the redirection of the stream in the callbacks. The return value is a struct with a **char** *ptr field that contains the handle value returned by the fmimewriteopen callback, and **char** *id, **char** *type, and **char** *description fields with the optional MIME id, type, and description info.

Example client-side code in C:

```
struct soap *soap = soap_new1(SOAP_ENC_MTOM);
soap_post_check_mime_attachments(soap);
...
```

```
if (soap_call_ns__myMethod(soap, ...))
  soap_print_fault(soap, stderr); // an error occurred
else
{
  if (soap_check_mime_attachments(soap))    { // attachments are present, channel is still open
    {
      do
      {
        ... // get data 'handle' from SOAP response and pass to callbacks
        ... // set the fmime callbacks, if needed
        struct soap_multipart *content = soap_get_mime_attachment(soap, (void*)handle);
        printf("Received attachment with id=%s and type=%s\n", content->id?content->id:"",
content->type?content->type:"");
      } while (content);
      if (soap->error)
        soap_print_fault(soap, stderr);
    }
  }
}
...
soap_destroy(soap);
soap_end(soap);
soap_free(soap); // detach and free context
```

The server-side service operations are implemented as usual, but with additional checks for MTOM/MIME attachments:

```
struct soap *soap = soap_new1(SOAP_ENC_MTOM);
soap_post_check_mime_attachments(soap);
...
soap_serve(soap);
...
int ns__myMethod(struct soap *soap, ...)
{   ... // server-side processing logic
  if (soap_check_mime_attachments(soap))    { // attachments are present, channel is still open
    {
      do
      {
        ... // get data 'handle' from SOAP request and pass to callbacks
        ... // set the fmime callbacks, if needed
        struct soap_multipart *content = soap_get_mime_attachment(soap, (void*)handle);
        printf("Received attachment with id=%s and type=%s\n", content->id?content->id:"",
content->type?content->type:"");
      } while (content);
      if (soap->error)
        return soap->error;
    }
  }
  ... // server-side processing logic
  return SOAP_OK;
}
```

16.5 Streaming Chunked MTOM/MIME

gSOAP automatically handles inbound chunked MTOM/MIME attachments (streaming or non-streaming). To transmit outbound MTOM/MIME attachments, the attachment sizes MUST be determined in advance to calculate HTTP message length required to stream MTOM/MIME over HTTP. However, gSOAP also supports the transmission of outbound chunked MTOM/MIME attachments without prior determination of MTOM/MIME attachment sizes when certain conditions are met. These conditions require either non-HTTP transport (use the output-mode SOAP_ENC_XML flag), or chunked HTTP transport (use the output-mode SOAP_IO_CHUNK flag). You can also use the SOAP_IO_STORE flag (which is also used automatically with compression to determine the HTTP content length header) but that cancels the benefits of streaming MTOM/MIME.

To stream chunked MTOM/MIME, set the __size field of an attachment to zero and enable HTTP chunking. The MTOM/MIME fmimeread callback then fetches data in chunks of any size between 1 and the value of the len argument. The fmimeread callback should return 0 upon reaching the end of the data stream.

17 XML Validation

The gSOAP XML parser applies basic rules to validate content. Constraints are not automatically verified unless explicitly set using flags. This helps to avoid interoperability problems with toolkits that do not strictly enforce validation rules. In addition, we cannot always use strict validation for SOAP RPC encoded messages, since SOAP RPC encoding adopts a very loose serialization format.

Validation constraints are enabled with the SOAP_XML_STRICT input mode flag set, e.g. with soap_set_imode(soap, SOAP_XML_STRICT) or soap_new(SOAP_XML_STRICT), see Section 9.12 for the complete list of flags.

17.1 Occurrence Constraints

17.1.1 Default Values

Default values can be defined for optional elements and attributes, which means that the default value will be used when the element or attribute value is not present in the parsed XML. See also Section 7.5.7 and examples in subsequent subsections below.

Default values must be primitive types, integer, float, string, etc. Default values can be specified for struct and class members, as shown in the example below:

```
struct ns__MyRecord
{
  int n = 5; // optional element with default value 5
  char *name = "none"; // optional element with default value "none"
  @enum ns__color { RED, WHITE, BLUE } color = RED; // optional attribute with default value
RED
};
```

Upon deserialization of absent data, these members will be set accordingly. When classes are instantiated with soap_new_ClassName the instance will be initialized with default values.

17.1.2 Elements with minOccurs and maxOccurs Restrictions

To force the validation of minOccurs and maxOccurs contraints the SOAP_XML_STRICT input mode flag must be set. The minOccurs and maxOccurs constraints are specified for fields of a struct and members of a class in a header file using the following syntax:

Type fieldname [minOccurs[:maxOccurs]] [= value]

The minOccurs and maxOccurs values must be integer literals. A default value can be provided. When minOccurs is not specified, it is assumed to be zero.

For example

```
struct ns__MyRecord
{
    int n = 5; // element with default value 5, minOccurs=0, maxOccurs=1
    int m 1; // element with minOccurs=1
    int __size 0:10; // sequence ¡item¿ with minOccurs=0, maxOccurs=10
    int *item;
    std::vector<double> nums 2; // sequence ¡nums¿ with minOccurs=2, maxOccurs=unbounded
};
struct arrayOfint
{
    int *__ptr 1:100; // minOccurs=1, maxOccurs=100
    int size;
};
```

Pointer-based struct fields and class members are allowed to be nillable when minOccurs is zero.

17.1.3 Required and Prohibited Attributes

Similar to the minOccurs and maxOccurs annotations defined in the previous section, attributes in a struct or class can be annotated with occurrence constraints to make them optional (0), required (1), or prohibited (0:0). Default values can be assigned to optional attributes.

For example

```
struct ns__MyRecord
{
    @int m 1; // required attribute (occurs at least once)
    @int n = 5; // optional attribute with default value 5
    @int o 0; // optional attribute (may or may not occur)
    @int p 0:0; // prohibited attribute
};
```

Remember to set the SOAP_XML_STRICT input mode flag to enable the validation of attribute occurrence constraints.

17.2   Value Constraints

17.2.1   Data Length Restrictions

A schema simpleType is defined with a **typedef** by taking a base primitive to defined a derived simpleType. For example:

typedef int time_ _seconds;

This defines the following schema type in time.xsd:

```
<simpleType name="seconds">
  <restriction base="xsd:int"/>
</simpleType>
```

A complexType with simpleContent is defined with a wrapper struct/class:

struct time_ _date
{
   char *_ _item; // some custom format date (restriction of string)
   @enum time_ _zone { EST, GMT, ... } zone;
}

This defines the following schema type in time.xsd:

```
<complexType name="date">
  <simpleContent>
    <extension base="xsd:string"/>
  </simpleContent>
  <attribute name="zone" type="time:zone" use="optional"/>
</complexType> <simpleType name="zone">
  <restriction base="xsd:string">
    <enumeration value="EST"/>
    <enumeration value="GMT"/>
    ...
  </restriction>
</simpleType>
```

Data value length constraints of simpleTypes and complexTypes with simpleContent are defined as follows.

typedef char *ns_ _string256 0:256; // simpleType restriction of string with max length 256 characters
typedef char *ns_ _string10 10:10; // simpleType restriction of string with length of 10 characters
typedef std::string *ns_ _string8 8; // simpleType restriction of string with at least 8 characters
struct ns_ _data // simpleContent wrapper
{
   char *_ _item :256; // simpleContent with at most 256 characters
   @char *name 1; // required name attribute
};

```
struct time__date // simpleContent wrapper
{
  char *__item :100;
  @enum time___zone { EST, GMT, ... } zone = GMT;
}
```

Remember to set the SOAP_XML_STRICT input mode flag to enable the validation of value length constraints.

17.2.2   Value Range Restrictions

Similar to data length constraints for string-based data, integer data value range constraints of numeric simpleTypes and complexTypes with simpleContent are defined as follows.

```
typedef int ns__int10 0:10; // simpleType restriction of int 0..10
typedef LONG64 ns__long -1000000:1000000; // simpleType restriction of long64 -1000000..1000000
typedef float ns__float100 -100:100; // simpleType restriction of float -100..100
struct ns__data // simpleContent wrapper
{
  int __item 0:10; // simpleContent range 0..10
  @char *name 1; // required name attribute
};
```

Currently the value bounds must be integer valued. Therefore, floating point ranges are limited to integer bounds. This may change in future releases.

17.2.3   Pattern Restrictions

Patterns can be defined for simpleType content. However, patterns are currently not enforced in the validation process though possibly in future releases.

To associate a pattern with a simpleType, you can define a simpleType with a **typedef** and a pattern string:

```
typedef int time__second "[1-5]?[0-9]—60";
```

This defines the following schema type in time.xsd:

```
<simpleType name="second">
  <restriction base="xsd:int">
    <pattern value="[1-5]?[0-9]|60"/>
  </restriction>
</simpleType>
```

The pattern string MUST contain a valid regular expression.

A special case for C format string patterns is introduced in gSOAP 2.8.18. When `xs:totalDigits` and `xs:fractionDigits` are given in a XSD file, then a C format string is produced to output floating point values with the proper precision and scale. For example:

```
<simpleType name="ratio">
  <restriction base="xsd:float">
    <totalDigits value="5"/>
    <fractionDigits value="2"/>
  </restriction>
</simpleType>
```

produces:

typedef float time_ _ratio "%5.2f";

The format string is used to format the output the floating point value in XML.

17.3   Element and Attribute Qualified/Unqualified Forms

Struct, class, and union members represent elements and attributes that are automatically qualified or unqualified depending on the schema element and attribute default forms. The gSOAP engine always validates the prefixes of elements and attributes. When a namespace mismatch occurs, the element or attribute is not consumed which can lead to a validation error (unless the complexType is extensible or when SOAP_XML_STRICT is turned off).

See Section 10.3 for details on the the struct/class/union member identifier translation rules. Consider for example:

```
//gsoap ns schema elementForm: qualified
//gsoap ns schema attributeForm: unqualified
struct ns_ _record
{
  @char * type;
  char * name;
};
```

Here, the ns_ _record struct is serialized with qualified element name and unqualified attribute type:

```
<ns:record type="...">
  <ns:name>...</ns:name>
</ns:record>
```

The "colon notation" for struct/class/union member field names is used to override element and attribute qualified or unqualified forms. To override the form for individual members that represent elements and attributes, use a namespace prefix and colon with the member name:

```
//gsoap ns schema elementForm: qualified
//gsoap ns schema attributeForm: unqualified
struct ns_ _record
{
  @char * ns:type;
  char * :name;
};
```

where `name` is unqualified and `type` is qualified:

```
<ns:record ns:type="...">
  <name>...</name>
</ns:record>
```

The colon notation is a syntactic notation used only in the gSOAP header file syntax, it is not translated to the C/C++ output.

The colon notation does not avoid name clashes between members. For example:

```
struct x__record
{
  @char * name;
  char * x:name;
};
```

results in a redefinition error, since both members have the same name. To avoid name clashes, use a underscore suffix:

```
struct x__record
{
  @char * name;
  char * x:name_;
};
```

Not that the namespace prefix convention can be used instead:

```
struct x__record
{
  @char * name;
  char * x__name;
};
```

which avoids the name clash. However, the resulting schema is different since the last example generates a global `name` element definition that is referenced by the local element.

More specifically, the difference between the namespace prefix convention with double underscores and colon notation is that the namespace prefix convention generates schema element/attribute references to elements/attributes at the top level, while the colon notation only affects the local element/attribute namespace qualification by form overriding. This is best illustrated by an example:

```
struct x__record
{
  char * :name;
  char * x:phone;
  char * x__fax;
  char * y__zip;
};
```

which generates the following x.xsdschema:

```
<complexType name="record">
  <sequence>
    <element name="name" type="xsd:string" minOccurs="0" maxOccurs="1" nillable="true"
form="unqualified"/>
    <element name="phone" type="xsd:string" minOccurs="0" maxOccurs="1" nillable="true"
form="qualified"/>
    <element ref="x:fax" minOccurs="0" maxOccurs="1"/>
    <element ref="y:zip" minOccurs="0" maxOccurs="1"/>
  </sequence>
</complexType>
<element name="fax" type="xsd:string"/>
```

and the y.xsd schema defines contains:

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

18   SOAP/XML Over UDP

UDP is a simple, unreliable datagram protocol: UDP sockets are connectionless. UDP address formats are identical to those used by TCP. In particular UDP provides a port identifier in addition to the normal Internet address format. The UDP port space is separate from the TCP port space (i.e. a UDP port may not be "connected" to a TCP port). In addition broadcast packets may be sent (assuming the underlying network supports this) by using a reserved "broadcast address"; this address is network interface dependent.

Client-side messages with SOAP-over-UDP endpoint URLs (soap.udp://...) will be automatically transmitted as datagrams. Server-side applications should set the SOAP_IO_UDP mode flag to accept UDP requests, e.g. using soap_init1 or soap_set_mode.

The maximum message length for datagram packets is restricted by the buffer size SOAP_BUFLEN, which is 65536 by default, unless compiled with WITH_LEAN to support small-scale embedded systems. For UDP transport SOAP_BUFLEN must not exceed the maximum UDP packet size 65536 (the size of datagram messages is constrained by the UDP packet size $2^{16} = 65536$ as per UDP standard). You can use gzip compression to reduce the message size, but note that compressed SOAP-over-UDP is a gSOAP-specific feature because it is not part of the SOAP-over-UDP specification.

The SOAP-over-UDP specification relies on WS-Addressing. The wsa.h file in the import directory defines the WS-Addressing elements for client and server applications.

The gSOAP implementation conforms to the SOAP-over-UDP requirements:

- SOAP-over-UDP server endpoint URL format: *soap.udp://host:port/path*

- Support one-way message-exchange pattern (MEP) where a SOAP envelope is carried in a user datagram.

- Support request-response message-exchange pattern (MEP) where SOAP envelopes are carried in user datagrams.

- Support multicast transmission of SOAP envelopes carried in user datagrams.

- Support both SOAP 1.1 and SOAP 1.2 envelopes.

The following additional features are also available, but are not supported by the SOAP-over-UDP specification:

- Zlib/gzip message compression (compile -DWITH_GZIP).

- SOAP with DIME attachments over UDP.

- SOAP with MIME attachments (SwA) over UDP.

- Support for IPv6 (compile -DWITH_IPV6)

18.1   Using WS-Addressing with SOAP-over-UDP

A SOAP-over-UDP application MUST use WS-Addressing to control message delivery as per SOAP-over-UDP specification.

The wsa.h file in the import directory defines the WS-Addressing elements. To include the WS-Addressing elements in the SOAP Header for messaging, a **struct** SOAP_ENV_ _Header structure must be defined in your header file with the appropriate WS-Addressing elements. For example:

```
#import "wsa.h"
struct SOAP_ENV_ _Header
{
  mustUnderstand _wsa_ _MessageID wsa_ _MessageID 0;
  mustUnderstand _wsa_ _RelatesTo *wsa_ _RelatesTo 0;
  mustUnderstand _wsa_ _From *wsa_ _From 0;
  mustUnderstand _wsa_ _ReplyTo *wsa_ _ReplyTo 0;
  mustUnderstand _wsa_ _FaultTo *wsa_ _FaultTo 0;
  mustUnderstand _wsa_ _To wsa_ _To 0;
  mustUnderstand _wsa_ _Action wsa_ _Action 0;
};
```

We also included a //gsoap wsa schema import directive in the wsa.h file to enable the generation of WSDL specifications that import (instead of includes) the WS-Addressing elements. Note that the //gsoapopt w directive must not be present in your header file to enable WSDL generation.

One-way SOAP-over-UDP messages (see Section 7.3) should be declared to include the wsa:MessageID, wsa:To, and wsa:Action elements in the SOAP Header of the request message as follows:

```
//gsoap ns service method-header-part: sendString wsa_ _MessageID
//gsoap ns service method-header-part: sendString wsa_ _To
//gsoap ns service method-header-part: sendString wsa_ _Action
int ns_ _sendString(char *str, void);
```

Request-response SOAP-over-UDP messages should be declared to include the wsa:MessageID, wsa:To, wsa:Action, and wsa:ReplyTo elements in the SOAP Header of the request message, and the the wsa:MessageID, wsa:To, wsa:Action, and wsa:RelatesTo elements in the SOAP Header of the response message:

```
//gsoap ns service method-header-part: echoString wsa__MessageID
//gsoap ns service method-header-part: echoString wsa__To
//gsoap ns service method-header-part: echoString wsa__Action
//gsoap ns service method-input-header-part: sendString wsa__ReplyTo
//gsoap ns service method-output-header-part: echoString wsa__RelatesTo
int ns__echoString(char *str, char **res);
```

For the content requirements of these elements, please consult the SOAP-over-UDP specification and/or read the next sections explaining SOAP-over-UDP unicast, multicast, one-way, and request-response client and server applications.

18.2   Client-side One-way Unicast

This example assumes that the gSOAP header file includes the SOAP Header with WS-Addressing elements and the ns__sendString function discussed in Section 18.1

```
struct soap soap;
struct SOAP_ENV__Header header; // the SOAP Header
soap_init(&soap);
soap.send_timeout = 1; // 1s timeout
soap_default_SOAP_ENV__Header(&soap, &header); // init SOAP Header
header.wsa__MessageID = "message ID";
header.wsa__To = "server URL";
header.wsa__Action = "server action";
soap.header = &header; // bind the SOAP Header for transport
// Send the message over UDP:
if (soap_send_ns__echoString(&soap, "soap.udp://...", NULL, "hello world!"))
   soap_print_fault(&soap, stderr); // report error
soap_end(&soap); // cleanup
soap_destroy(&soap); // cleanup
soap_done(&soap); // close connection (should not use soap struct after this)
```

18.3   Client-side One-way Multicast

This example is similar to the one-way unicast example discussed above, but uses a broadcast address and the SO_BROADCAST socket option:

```
struct soap soap;
struct SOAP_ENV__Header header; // the SOAP Header
in_addr_t addr = inet_addr("1.2.3.4"); // optional
soap_init(&soap);
soap.send_timeout = 1; // 1s timeout
soap.connect_flags = SO_BROADCAST; // required for broadcast
```

```
soap.ipv4_multicast_if = &addr; // optional for IPv4: see setsockopt IPPROTO_IP IP_MULTICAST_IF
soap.ipv6_multicast_if = addr; // optional for IPv6: multicast sin6_scope_id
soap.ipv4_multicast_ttl = 1; // optional, see setsockopt IPPROTO_IP, IP_MULTICAST_TTL
soap_default_SOAP_ENV__Header(&soap, &header); // init SOAP Header
header.wsa__MessageID = "message ID";
header.wsa__To = "server URL";
header.wsa__Action = "server action";
soap.header = &header; // bind the SOAP Header for transport
// Send the message over UDP to a broadcast address:
if (soap_send_ns__echoString(&soap, "soap.udp://...", NULL, "hello world!"))
  soap_print_fault(&soap, stderr); // report error
soap_destroy(&soap); // cleanup
soap_end(&soap); // cleanup
soap_done(&soap); // close connection (should not use soap struct after this)
```

Please refer to the socket options for IPPROTO_IP IP_MULTICAST_IF to specify the default interface for multicast datagrams to be sent from. This is a **struct** in_addr (in_addr_t for sin6_scope_id) interface value. Otherwise, the default interface set by the system administrator will be used (if any).

Please refer to the socket options for IPPROTO_IP IP_MULTICAST_TTL to limit the lifetime of the packet. Multicast datagrams are sent with a default value of 1, to prevent them to be forwarded beyond the local network. This parameter can be set between 1 to 255.

18.4   Client-side Request-Response Unicast

This example assumes that the gSOAP header file includes the SOAP Header with WS-Addressing elements and the ns__echoString function discussed in Section 18.1

```
struct soap soap;
struct SOAP_ENV__Header header; // the SOAP Header
struct wsa__EndpointReferenceType replyTo; // (anonymous) reply address
char *res; // server response
soap_init(&soap);
soap.send_timeout = 1; // 1s timeout
soap.recv_timeout = 1; // 1s timeout
soap_default_SOAP_ENV__Header(&soap, &header); // init SOAP Header
soap_default_wsa__EndpointReferenceType(&soap, &replyTo); // init reply address
replyTo.Address = "http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous";
header.wsa__MessageID = "message ID";
header.wsa__To = "server URL";
header.wsa__Action = "server action";
header.wsa__ReplyTo = &replyTo;
soap.header = &header; // bind the SOAP Header for transport
// Send and receive messages over UDP:
if (soap_call_ns__echoString(&soap, "soap.udp://...", NULL, "hello world!", &res))
{
  if (soap.error == SOAP_EOF && soap.errnum == 0)
    // Timeout: no response from server (message already delivered?)
  else
```

```
        soap_print_fault(&soap, stderr);
}
else
    // UDP server response is stored in 'res'
// check SOAP header received, if applicable
check_header(&soap.header);
soap_destroy(&soap); // cleanup
soap_end(&soap); // cleanup
soap_done(&soap); // close connection (should not use soap struct after this)
```

18.5 Client-side Request-Response Multicast

This example is similar to the request-response unicast example discussed above, but uses a broadcast address and the SO_BROADCAST socket option. Because we expect to receive multiple responses, we also need to use separate request-response messages to send one request and consume multiple responses. In this example we defined a bcastString request and a bcastStringResponse response message, which are essentially declared as one-way messages in the header file:

```
//gsoap ns service method-header-part: bcastString wsa__MessageID
//gsoap ns service method-header-part: bcastString wsa__To
//gsoap ns service method-header-part: bcastString wsa__Action
//gsoap ns service method-header-part: bcastString wsa__ReplyTo
int ns__bcastString(char *str, void);
//gsoap ns service method-header-part: bcastStringResponse wsa__MessageID
//gsoap ns service method-header-part: bcastStringResponse wsa__To
//gsoap ns service method-header-part: bcastStringResponse wsa__Action
//gsoap ns service method-header-part: bcastStringResponse wsa__RelatesTo
int ns__bcastStringResponse(char *res, void);
```

To obtain response one-way operations, use the wsdl2h -b option.

The client code includes a loop to receive response messages until a timeout occurs:

```
struct soap soap;
struct SOAP_ENV__Header header;
struct wsa__EndpointReferenceType replyTo;
char *res;
soap_init(&soap);
soap.connect_flags = SO_BROADCAST;
soap.send_timeout = 1; // 1s timeout
soap.recv_timeout = 1; // 1s timeout
soap_default_SOAP_ENV__Header(&soap, &header);
soap.header = &header;
soap_default_wsa__EndpointReferenceType(&soap, &replyTo);
replyTo.Address = "http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous";
header.wsa__MessageID = "message ID";
header.wsa__To = "server URL";
header.wsa__Action = "server action";
header.wsa__ReplyTo = &replyTo;
if (soap_send_ns__bcastString(&soap, "soap.udp://...", NULL, "hello world!"))
```

```
      soap_print_fault(&soap, stderr);
   else
   {
      for (;;)
      {
         if (soap_recv_ns__bcastStringResponse(&soap, &res))
            break;
         // Got response 'res' from a server
      }
      if (soap.error == SOAP_EOF && soap.errnum == 0)
         // Timeout: no more messages received
      else
         soap_print_fault(&soap, stderr);
   }
   soap_destroy(&soap); // cleanup
   soap_end(&soap); // cleanup
   soap_done(&soap); // close connection (should not use soap struct after this)
```

Note that a server for the bcastString does not need to use two-one way messages. Thus, multicast request-response message pattern can be declared and implemented as request-response operations at the server side.

18.6 SOAP-over-UDP Server

The following example code illustrates a SOAP-over-UDP server for one-way sendString and request-response echoString messages. This example assumes that the gSOAP header file includes the SOAP Header with WS-Addressing elements and the ns__echoString function discussed in Section 18.1.

```
int main()
{
   struct soap soap;
   soap_init1(&soap, SOAP_IO_UDP); // must set UDP flag
   // bind to host (NULL=current host) and port:
   if (!soap_valid_socket(soap_bind(&soap, host, port, 100)))
   {
      soap_print_fault(&soap, stderr);
      exit(1);
   }
   for (;;)
   {
      if (soap_serve(&soap))
         soap_print_fault(&soap, stderr); // report the problem
      soap_destroy(&soap);
      soap_end(&soap);
   }
   soap_done(&soap); // close connection
}
int ns__echoString(struct soap *soap, char *str, char **res)
{
   if (!soap->header)
```

below

```
      return soap_sender_fault(soap, "No SOAP header", NULL);
    if (!soap->header->wsa__MessageID)
      return soap_sender_fault(soap, "No WS-Addressing MessageID", NULL);
    soap->header->wsa__RelatesTo = (struct wsa__Relationship*)soap_malloc(soap, sizeof(struct
wsa__Relationship));
    soap_default_wsa__Relationship(soap, soap->header->wsa__RelatesTo);
    soap->header->wsa__RelatesTo->__item = soap->header->wsa__MessageID;
    // must check for duplicate messages
    if (check_received(soap->header->wsa__MessageID))
    {
      // Request message already received
      return SOAP_STOP; // don't return response
    }
    if (!soap->header->wsa__ReplyTo || !soap->header->wsa__ReplyTo->Address)
      return soap_sender_fault(soap, "No WS-Addressing ReplyTo address", NULL);
    soap->header->wsa__To = soap->header->wsa__ReplyTo->Address;
    soap->header->wsa__MessageID = soap_strdup(soap, soap_int2s(soap, id_count++)) ;
    soap->header->wsa__Action = "http://genivia.com/udp/echoStringResponse";
    *res = str;
    return SOAP_OK;
}
int ns__sendString(struct soap *soap, char *str)
{
    if (!soap->header)
      return SOAP_STOP;
    if (!soap->header->wsa__MessageID)
      return SOAP_STOP;
    // must check for duplicate messages
    if (check_received(soap->header->wsa__MessageID))
      return SOAP_STOP;
    return SOAP_OK;
}
int ns__sendStringResponse(struct soap *soap, char *res)
{ return SOAP_NO_METHOD; } // we don't expect to serve this message
```

The server binds to a host and port and accepts messages in a tight sequential loop. Because UDP does not have the equivalent of an accept the messages cannot be dispatched to threads, the soap_serve waits for a message and immediately accepts it. You can use a receive timeout to make soap_serve non-blocking.

To obtain response one-way operations from a WSDL, use the wsdl2h -b option. This produces additional one-way operations to support asynchronous handling of response messages in the same way requests are handled.

18.7   SOAP-over-UDP Multicast Receiving Server

For UDP multicast support, follow the suggestions in Section 18.6 and change the initialization parts of the code to enable UDP multicast port binding by to telliing the kernel which multicast groups you are interested in:

```
int main()
{
  struct soap soap;
  struct ip_mreq mcast;
  soap_init1(&soap, SOAP_IO_UDP);    if (!soap_valid_socket(soap_bind(&soap, host, port, 100)))
  {
    soap_print_fault(&soap, stderr);
    exit(1);
  }
  mcast.imr_multiaddr.s_addr = inet_addr("put IP multicast address of group here");
  mcast.imr_interface.s_addr = htonl(INADDR_ANY);
  if (setsockopt(soap.master, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mcast, sizeof(mcast))¡0)
    ... error ...
```

19 Advanced Features

19.1 Internationalization

gSOAP uses regular strings by default. Regular strings cannot be used to hold UCS characters outside of the character range [1,255]. gSOAP can handle wide-character content in two ways. First, applications can utilize wide-character strings (wchar_t*) instead of regular strings to store wide-character content. For example, the xsd:string string schema type can be declared as a wide-character string and used subsequently:

```
typedef wchar_t *xsd__string;
...
int ns__myMethod(xsd__string input, xsd__string *output);
```

Second, regular strings can be used to hold wide-character content in UTF-8 format. This is accomplished with the SOAP_C_UTFSTRING flag (for both input/output mode), see Section 9.12. With this flag set, gSOAP will deserialize XML into regular strings in UTF-8 format. An application is responsible for filling regular strings with UTF-8 content to ensure that strings can be correctly serialized XML. Third, the SOAP_C_MBSTRING flag (for both input/output mode) can be used to activate multibyte character support. Multibyte support depends on the locale settings for dealing with extended natural language encodings.

Both regular strings and wide-character strings can be used together within an application. For example, the following header file declaration introduces two string schema types:

```
typedef wchar_t *xsd__string;
typedef char *xsd__string_; // trailing '_' avoids name clash
...
int ns__myMethod(xsd__string input, xsd__string_ *output);
```

The input string parameter is a wide-character string and the output string parameter is a regular string. The regular string has UCS character content in the range [1,255] unless the SOAP_C_UTFSTRING flag is set. With this flag, the string has UTF-8 encoded content.

Please consult the UTF-8 specification for details on the UTF-8 format. Note that the ASCII character set [1-127] is a subset of UTF-8. Therefore, with the SOAP_C_UTFSTRING flag set, strings may hold ASCII character data and UTF-8 extensions.

19.2 Customizing the WSDL and Namespace Mapping Table File Contents With gSOAP Directives

A header file can be augmented with directives for the gSOAP soapcpp2 tool to automatically generate customized WSDL and namespace mapping tables contents. The WSDL and namespace mapping table files do not need to be modified by hand (Sections 7.2.9 and 10.4). In addition, the sample SOAP/XML request and response files generated by the compiler are valid provided that XML Schema namespace information is added to the header file with directives so that the gSOAP soapcpp2 compiler can produce example SOAP/XML messages that are correctly namespace qualified. These compiler directive are specified as //-comments. (Note: blanks can be used anywhere in the directive, except between // and gsoap.)

Three directives are currently supported that can be used to specify details associated with namespace prefixes used by the service operation names in the header file. To specify the name of a Web Service in the header file, use:

//gsoap *namespace-prefix* service name: *service-name*

where *namespace-prefix* is a namespace prefix used by identifiers in the header file and *service-name* is the name of a Web Service (only required to create new Web Services). The name may be followed by text up to the end of the line which is incorporated into the WSDL service documentation. Alternatively, the service documentation can be provided with the directive below.

To specify the name of the WSDL definitions in the header file, use:

//gsoap *namespace-prefix* service definitions: *definitions-name*

where *namespace-prefix* is a namespace prefix used by identifiers in the header file and *definitions-name* is the name of the WSDL definitions. By default, the WSDL definitions name is the same as the service name.

To specify the documentation of a Web Service in the header file, use:

//gsoap *namespace-prefix* service documentation: *text*

where *namespace-prefix* is a namespace prefix used by identifiers in the header file and *text* is the documentation text up to the end of the line. The text is incorporated into the WSDL service documentation.

To specify the portType of a Web Service in the header file, use:

//gsoap *namespace-prefix* service portType: *portType-name*

or just

> //gsoap *namespace-prefix* service type: *portType-name*

or using WSDL 2.0 terms

> //gsoap *namespace-prefix* service interface: *portType-name*

where *namespace-prefix* is a namespace prefix used by identifiers in the header file and *portType-name* is the portType name of the WSDL service portType.

To specify the port name of a Web Service in the header file, use:

> //gsoap *namespace-prefix* service portName: *port-name*

where *namespace-prefix* is a namespace prefix used by identifiers in the header file and *port-name* is the name of the WSDL service port element. By default, the port name is the same as the service name.

To specify the binding name of a Web Service in the header file, use:

> //gsoap *namespace-prefix* service binding: *binding-name*

where *namespace-prefix* is a namespace prefix used by identifiers in the header file and *binding-name* is the binding name of the WSDL service binding element. By default, the binding name is the same as the service name.

To specify the binding's transport protocol of a Web Service in the header file, use:

> //gsoap *namespace-prefix* service transport: *transport-URL*

where *namespace-prefix* is a namespace prefix used by identifiers in the header file and *transport-URL* is the URL of the transport protocol such as http://schemas.xmlsoap.org/soap/http for HTTP. HTTP transport is assumed by default.

To specify the location (or port endpoint) of a Web Service in the header file, use:

> //gsoap *namespace-prefix* service location: *URL*

or alternatively

> //gsoap *namespace-prefix* service endpoint: *URL*

or

> //gsoap *namespace-prefix* service port: *URL*

where *URL* is the location of the Web Service (only required to create new Web Services). The *URL* specifies the path to the service executable (so *URL/service-executable* is the actual location of the executable when declared).

To specify the name of the executable of a Web Service in the header file, use:

220

```
//gsoap namespace-prefix service executable: executable-name
```

where *executable-name* is the name of the executable of the Web Service.

When doc/literal encoding is required for the entire service, the service encoding can be specified in the header file as follows:

```
//gsoap namespace-prefix service encoding: literal
```

or when the `SOAP-ENV:encodingStyle` attribute is different from the SOAP 1.1/1.2 encoding style:

```
//gsoap namespace-prefix service encoding: encoding-style
```

To specify the namespace URI of a Web Service in the header file, use:

```
//gsoap namespace-prefix service namespace: namespace-URI
```

where *namespace-URI* is the URI associated with the namespace prefix.

In addition, the schema namespace URI can be specified in the header file:

```
//gsoap namespace-prefix schema namespace: namespace-URI
```

where *namespace-URI* is the schema URI associated with the namespace prefix. If present, it defines the schema-part of the generated WSDL file and the URI in the namespace mapping table. This declaration is useful when the service declares its own data types that need to be associated with a namespace. Furthermore, the header file for client applications do not need the full service details and the specification of the schema namespaces for namespace prefixes suffices. In addition, a second namespace can be defined that is only used to match the namespaces of inbound XML:

```
//gsoap namespace-prefix schema namespace2: namespace-URI-pattern
```

If the first namespace does not match the inbound parsed XML, then the second will be tried. This pattern may contain '*' multichar wildcards and '-' single chard wildcards. This allows two or more namespace versions to be handled by the same namespace prefix.

The directive above specifies a new schema and the gSOAP soapcpp2 compiler generates a schema files (.xsd) file for the schema. An existing schema namespace URI can be imported with:

```
//gsoap namespace-prefix schema import: namespace-URI
```

where *namespace-URI* is the schema URI associated with the namespace prefix. gSOAP does not produce XML Schema files for imported schemas and imports the schema namespaces in the generated WSDL file.

A schema namespace URI can be imported from a location with:

```
//gsoap namespace-prefix schema namespace: namespace-URI
//gsoap namespace-prefix schema import: schema-location
```

The elementFormDefault and attributeFormDefault qualification of a schema can be defined with:

> //gsoap *namespace-prefix* schema elementForm: qualified
> //gsoap *namespace-prefix* schema attributeForm: qualified

or:

> //gsoap *namespace-prefix* schema elementForm: unqualified
> //gsoap *namespace-prefix* schema attributeForm: unqualified

A shortcut to define the default qualification of elements and attributes of a schema:

> //gsoap *namespace-prefix* schema form: qualified

or:

> //gsoap *namespace-prefix* schema form: unqualified

To include `xsi:type` attributes in the runtime XML element output for specific schemas, use:

> //gsoap *namespace-prefix* schema typed: yes

Note that soapcpp2 -t enables `xsi:type` for all elements in the runtime XML output.

To document a data type, use:

> //gsoap *namespace-prefix* schema type-documentation: *type-name* //text

where *type-name* is the unqualified name of the data type and *text* is a line of text terminated by a newline. Do not use any XML reserved characters in *text* such as < and >. Use well-formed XML and XHTML markup instead. For example:

> //gsoap ns schema type-documentation: tdata stores transaction
> data
> **class** ns_ _tdata
> { ... }

To document a data type's fields and members, use:

> //gsoap *namespace-prefix* schema type-documentation: *type-name::field* //text

where *type-name* is the unqualified name of the data type, *field* is a field, member, or enum name, and *text* is a line of text terminated by a newline. Do not use any XML reserved characters in *text* such as < and >. Use well-formed XML and XHTML markup instead. For example:

> //gsoap ns schema type-documentation: tdata::id the transaction number
> //gsoap ns schema type-documentation: tdata::state transaction state
> //gsoap ns schema type-documentation: tstate::INIT initial state
> //gsoap ns schema type-documentation: tstate::DONE final state

```
class ns__tdata
{ @int id;
   enum ns__tstate { INIT, DONE } state;
   ...
}
```

The documentation form above can also be used to document SOAP/XML message parts in the generated WSDL. For the type-name use the function name. For the field names, you can use the function name and/or the function argument names.

To document a method, use:

//gsoap *namespace-prefix* service method-documentation: *method-name* //*text*

where *method-name* is the unqualified name of the method and *text* is a line of text terminated by a newline. Do not use any XML reserved characters in *text* such as < and >. Use well-formed XML and XHTML markup instead. For example:

```
//gsoap ns service method-documentation: getQuote returns a <i>stock quote</i>
int ns__getQuote(char *symbol, float &_result);
```

To specify the SOAP Action for a SOAP method, use:

//gsoap *namespace-prefix* service method-action: *method-name action*

where *method-name* is the unqualified name of the method and *action* is a string without spaces and blanks (the string can be quoted when preferred). For example:

```
//gsoap ns service method-action: getQuote ""
int ns__getQuote(char *symbol, float &_result);
```

Or, alternatively for the input action (part of the request):

```
//gsoap ns service method-input-action: getQuote ""
int ns__getQuote(char *symbol, float &_result);
```

To specify the HTTP "location" of REST methods to a perform POST/GET/PUT action, use:

//gsoap *namespace-prefix* service method-action: *method-name action*

where *method-name* is the unqualified name of the method and *action* is a string without spaces and blanks (the string can be quoted when preferred). This directive requires that the protocol: directive for this method is set to HTTP, POST, GET, or PUT.

A response action and fault action are defined by:

//gsoap *namespace-prefix* service method-output-action: *method-name action* //gsoap *namespace-prefix* service method-fault-action: *method-name action*

To override the SOAP or REST protocol of an operation (SOAP by default), use:

//gsoap *namespace-prefix* service method-protocol: *method-name protocol*

where *protocol* is one of

SOAP	default SOAP transport (supports 1.1 and 1.2)
SOAP1.1	SOAP 1.1 only
SOAP1.2	SOAP 1.2 only
SOAP-GET	one-way SOAP with HTTP GET
SOAP1.1-GET	one-way SOAP 1.1 with HTTP GET
SOAP1.2-GET	one-way SOAP 1.1 with HTTP GET
HTTP	REST HTTP (POST or one-way PUT)
POST	REST HTTP POST
GET	one-way REST HTTP GET
PUT	one-way REST HTTP PUT
DELETE	REST HTTP DELETE

When document style is preferred for a particular service method, use:

//gsoap *namespace-prefix* service method-style: *method-name* document

When SOAP RPC encoding is required for a particular service method, use:

//gsoap *namespace-prefix* service method-encoding: *method-name* encoded

When literal encoding is required for a particular service method, use:

//gsoap *namespace-prefix* service method-encoding: *method-name* literal

or when the `SOAP-ENV:encodingStyle` attribute is different from the SOAP 1.1/1.2 encoding style, use:

//gsoap *namespace-prefix* service method-encoding: *method-name encoding-style*

When SOAP RPC encoding is required for a particular service method response when the request message is literal, use:

//gsoap *namespace-prefix* service method-response-encoding: *method-name* encoded

When literal encoding is required for a particular service method response when the request message is encoded, use:

//gsoap *namespace-prefix* service method-response-encoding: *method-name* literal

or when the `SOAP-ENV:encodingStyle` attribute is different from the SOAP 1.1/1.2 encoding style, use:

//gsoap *namespace-prefix* service method-response-encoding: *method-name encoding-style*

Note that the method-response-encoding is set to the value of method-encoding by default.

When header processing is required, each method declared in the WSDL should provide a binding to the parts of the header that may appear as part of a method request message. Such a binding is given by:

```
//gsoap namespace-prefix service method-header-part: method-name header-part
```

For example:

```
struct SOAP_ENV__Header
{
    char *h__transaction;
    struct UserAuth *h__authentication;
};
```

Suppose method ns__login uses both header parts (at most), then this is declared as:

```
//gsoap ns service method-header-part: login h__transaction
//gsoap ns service method-header-part: login h__authentication
int ns__login(...);
```

Suppose method ns__search uses only the first header part, then this is declared as:

```
//gsoap ns service method-header-part: search h__transaction
int ns__search(...);
```

Note that the method name and header part names MUST be namespace qualified. The headers MUST be present in all operations that declared the header parts.

To specify the header parts for the method input (method request message), use:

```
//gsoap namespace-prefix service method-input-header-part: method-name header-part
```

Similarly, to specify the header parts for the method output (method response message), use:

```
//gsoap namespace-prefix service method-output-header-part: method-name header-part
```

The declarations above only affect the WSDL. For example:

```
struct SOAP_ENV__Header
{
    char *h__transaction;
    struct UserAuth *h__authentication;
};
//gsoap ns service method-input-header-part: login h__authentication
//gsoap ns service method-input-header-part: login h__transaction
//gsoap ns service method-output-header-part: login h__transaction
int ns__login(...);
```

The headers MUST be present in all operations that declared the header parts.

To specify MIME attachments for the method input and output (method request and response messages), use:

//gsoap *namespace-prefix* service method-mime-type: *method-name mime-type*

You can repeat this directive for all multipartRelated MIME attachments you want to associate with the method.

To specify MIME attachments for the method input (method request message), use:

//gsoap *namespace-prefix* service method-input-mime-type: *method-name mime-type*

Similarly, to specify MIME attachments for the method output (method response message), use:

//gsoap *namespace-prefix* service method-output-mime-type: *method-name mime-type*

You can repeat these directives for all multipartRelated MIME attachments you want to associate with the method.

19.2.1 Example

The use of directives is best illustrated with an example. The example uses a hypothetical stock quote service and exchange rate service, actual services such as these are available for free on the web.

```
//gsoap ns1 service namespace: urn:GetQuote
int ns1__getQuote(char *symbol, float &result);

//gsoap ns2 service namespace: urn:CurrencyExchange
int ns2__getRate(char *country1, char *country2, float &result);

//gsoap ns3 service name: quotex
//gsoap ns3 service style: rpc
//gsoap ns3 service encoding: encoded
//gsoap ns3 service port: http://www.mydomain.com/quotex.cgi
//gsoap ns3 service namespace: urn:quotex
int ns3__getQuote(char *symbol, char *country, float &result);
```

The quotex.h example is a new Web Service created by combining two existing Web Services: a Stock Quote service and a Currency Exchange service.

Namespace prefix ns3 is used for the new quotex Web Service with namespace URI urn:quotex, service name quotex, and endpoint port http://www.mydomain.com/quotex.cgi.

Since the new Web Service invokes the ns1__getQuote and ns2__getRate service operations, the service namespaces and other details such as style and encoding of these methods are given by directives. After invoking the gSOAP soapcpp2 tool on the quotex.h header file:

```
> soapcpp2 quotex.h
```

the WSDL of the new quotex Web Service is saved as quotex.wsdl. Since the service name, endpoint port, and namespace URI were provided in the header file, the generated WSDL file can be published together with the compiled Web Service installed as a CGI application.

The namespace mapping table for the quotex.cpp Web Service implementation is saved as quotex.nsmap. This file can be directly included in quotex.cpp instead of specified by hand in the source of quotex.cpp:

```
#include "quotex.nsmap"
```

The automatic generation and inclusion of the namespace mapping table requires compiler directives for **all** namespace prefixes to associate each namespace prefix with a namespace URI. Otherwise, namespace URIs have to be manually added to the table (they appear as http://tempuri.org).

19.3   Transient Data Types

There are situations when certain data types have to be ignored by gSOAP for the compilation of (de)marshalling routines. For example, in certain cases only a few members of a class or struct need not be (de)serialized, or the base class of a derived class should not be (de)serialized. Certain built-in classes such as ostream cannot be (de)serialized. Data parts that should be kept invisible to gSOAP are called "transient". Transient data types and transient struct/class members are declared with the **extern** keyword or are declared within [and] blocks in the header file. The **extern** keyword has a special meaning to the gSOAP soapcpp2 compiler and won't affect the generated codes. The special [and] block construct can be used with data type declarations and within **struct** and **class** declarations. The use of **extern** or [ ] achieve the same effect, but [ ] may be more convenient to encapsulate transient types in a larger part of the header file. The use of **extern** with **typedef** is reserved for the declaration of user-defined external (de)serializers for data types, see Section 19.5.

First example:

```
extern class ostream; // ostream can't be (de)serialized, but need to be declared to make it visible
to gSOAP
class ns__myClass
{ ...
   virtual void print(ostream &s) const; // need ostream here
   ...
};
```

Second example:

```
[
   class myBase // base class need not be (de)serialized
   { ... };
]
class ns__myDerived : myBase
{ ... };
```

Third example:

```
[ typedef int transientInt; ]
class ns_ _myClass
{
  int a; // will be (de)serialized
  [
  int b; // transient field
  char s[256]; // transient field
  ]
  extern float d; // transient field
  char *t; // will be (de)serialized
  transientInt *n; // transient field
  [
  virtual void method(char buf[1024]); // does not create a char[1024] (de)serializer
  ]
};
```

In this example, **class** ns_ _myClass has three transient fields: b, s, and n which will not be (de)serialized in SOAP. Field n is transient because the type is declared within a transient block. Pointers, references, and arrays of transient types are transient. The single class method is encapsulated within [and] to prevent gSOAP from creating (de)serializers for the **char**[1024] type. gSOAP will generate (de)serializers for all types that are not declared within a [and] transient block.

19.4   Serialization "as is" with Volatile Data Types

Volatile-declared data types in gSOAP are assumed to be part of an existing non-modifiable software package, such as a built-in library. It would not make sense to redefine the data types in a gSOAP header file. In certain cases it could also be problematic to have classes augmented with serializer methods. When you need to (de)serialize such data types "as is", you must declare them in a gSOAP header file and use the **volatile** qualifier.

Consider for example **struct** tm, declared in time.h. The structure may actually vary between platforms, but the tm structure includes at least the following fields:

```
volatile struct tm
{
  int tm_sec; /* seconds (0 - 60) */
  int tm_min; /* minutes (0 - 59) */
  int tm_hour; /* hours (0 - 23) */
  int tm_mday; /* day of month (1 - 31) */
  int tm_mon; /* month of year (0 - 11) */
  int tm_year; /* year - 1900 */
  int tm_wday; /* day of week (Sunday = 0) */
  int tm_yday; /* day of year (0 - 365) */
  int tm_isdst; /* is summer time in effect? */
  char *tm_zone; /* abbreviation of timezone name */
  long tm_gmtoff; /* offset from UTC in seconds */
};
```

Note that we qualified the structure **volatile** in the gSOAP header file to inform the gSOAP soapcpp2 compiler that it should not attempt to redeclare it. We can now readily serialize and deserialize the tm structure. The following program fragment serializes the local time stored in a tm structure to stdout:

```
struct soap *soap = soap_new();
...
time_t T = time(NULL);
struct tm *t = localtime(&T);
struct soap *soap = soap_new();
soap_write_tm(soap, t);
soap_destroy(soap);
soap_end(soap);
soap_free(soap); // detach and free context
```

It is also possible to serialize the tm fields as XML attributes using the @ qualifier, see Section 11.6.7.

If you must produce a schema file, say time.xsd, that defines an XML schema and namespace for the tm struct, you can add a **typedef** declaration to the header file:

```
typedef struct tm time__struct_tm;
```

We used the **typedef** name time__struct_tm rather than time__tm, because a schema name clash will occur with the latter since taking off the time prefix will result in the same name being used.

Classes should be declared volatile to prevent modification of these classes by ithe gSOAP soapcpp2 source code output. Note that gSOAP adds serialization methods to classes to support polymorphism. However, this is a problem when you can't modify class declarations because they are part of a non-modifiable software package. The solution is to declare these classes **volatile**, similar to the tm structure example illustrated above. You can also use a **typedef** to associate a schema with a class.

19.5   How to Declare User-Defined Serializers and Deserializers

Users can declare their own (de)serializers for specific data types instead of relying on the gSOAP-generated (de)serializers. To declare a external (de)serializer, declare a type with **extern typedef**. gSOAP will not generate the (de)serializers for the type name that is declared. For example:

```
extern typedef char *MyData;
struct Sample
{
  MyData s; // use user-defined (de)serializer for this field
  char *t; // use gSOAP (de)serializer for this field
};
```

The user is required to supply the following routines for each **extern typedef**'ed name $\underline{T}$:

```
void soap_serialize_T(struct soap *soap, const T *a)
void soap_default_T(struct soap *soap, T *a)
```

```
        void soap_out_T(struct soap *soap, const char *tag, int id, const T *a, const char *type)
        T *soap_in_T(struct soap *soap, const char *tag, T *a, const char *type)
```

The function prototypes can be found in soapH.h.

For example, the (de)serialization of MyData can be done with the following code:

```
        void soap_serialize_MyData(struct soap *soap, MyData *const*a)
        { } // no need to mark this node (for multi-ref and cycle detection)
        void soap_default_MyData(&soap, MyData **a)
        { *a = NULL }
        void soap_out_MyData(struct soap *soap, const char *tag, int id, MyData *const*a, const char
        *type)
        {
          soap_element_begin_out(soap, tag, id, type); // print XML beginning tag
          soap_send(soap, *a); // just print the string (no XML conversion)
          soap_element_end_out(soap, tag); // print XML ending tag
        }
        MyData **soap_in_MyData(struct soap *soap, const char *tag, MyData **a, const char *type)
        {
          if (soap_element_begin_in(soap, tag))
            return NULL;
          if (!a)
            a = (MyData**)soap_malloc(soap, sizeof(MyData*));
          if (soap->null)
            *a = NULL; // xsi:nil element
          if (*soap->type && soap_match_tag(soap, soap->type, type))
          {
            soap->error = SOAP_TYPE;
            return NULL; // type mismatch
          }
          if (*soap->href)
            a = (MyData**)soap_id_forward(soap, soap->href, a, SOAP_MyData, sizeof(MyData*))
          else if (soap->body)
          {
            char *s = soap_value(soap); // fill buffer
            *a = (char*)soap_malloc(soap, strlen(s)+1);
            strcpy(*a, s);
          }
          if (soap->body && soap_element_end_in(soap, tag))
            return NULL;
          return a;
```

More information on custom (de)serialization will be provided in this document or in a separate document in the future. The writing of the (de)serializer code requires the use of the low-level gSOAP API.

19.6 How to Serialize Data Without Generating XSD Type Attributes

gSOAP serializes data in XML with xsi:type attributes when the types are declared with namespace prefixes to indicate the schema type of the data contained in the elements. SOAP 1.1 and 1.2

requires `xsi:type` attributes in the presence of polymorphic data or when the type of the data cannot be deduced from the SOAP payload. The namespace prefixes are associated with the type names of **typedef**s (Section 11.3) for primitive data types, **struct**/class names, and **enum** names.

To prevent the output of these `xsi:type` attributes in the XML serialization, you can simply use type declarations that do not include these namespace prefixes. That is, don't use the **typedef**s for primitive types and use unqualified type names with **struct**s, **class**es, and enums.

However, there are two issues. Firstly, if you want to use a primitive schema type that has no C/C++ counterpart, you must declare it as a **typedef** name with a leading underscore, as in:

> **typedef char** *_xsd_ _date;

This will produce the necessary `xsd:date` information in the WSDL output by the gSOAP `soapcpp2` compiler. But the XML serialization of this type at run time won't include the `xsi:type` attribute. Secondly, to include the proper schema definitions in the WSDL produced by the gSOAP `soapcpp2` compiler, you should use qualified **struct**, **class**, and **enum** names with a leading underscore, as in:

> **struct** _ns_ _myStruct
> { ... };

This ensures that `myStruct` is associated with a schema, and therefore included in the appropriate schema in the generated WSDL. The leading underscore prevents the XML serialization of `xsi:type` attributes for this type in the SOAP/XML payload.

19.7 Function Callbacks for Customized I/O and HTTP Handling

gSOAP provides five callback functions for customized I/O and HTTP handling:

Callback (function pointer)

SOAP_SOCKET (*soap.fopen)(**struct** soap *soap, **const char** *endpoint, **const char** *host, **int** port)
Called from a client proxy to open a connection to a Web Service located at `endpoint`. Input parameters `host` and `port` are micro-parsed from `endpoint`. Should return a valid file descriptor, or SOAP_INVALID_SOCKET and soap->error set to an error code. Built-in gSOAP function: `tcp_connect`

int (*soap.fclose)(**struct** soap *soap)
Called by client proxy **multiple times**, to close a socket connection before a new socket connection is established and at the end of communications when the SOAP_IO_KEEPALIVE flag is not set and soap.keep_alive≠0 (indicating that the other party supports keep alive). Should return SOAP_OK, or a gSOAP error code. Built-in gSOAP function: `tcp_disconnect`

Callback (function pointer)

int (*soap.fget)(**struct** soap *soap)

Called by the main server loop upon an HTTP GET request. The SOAP_GET_METHOD error is returned by default. This callback can be used to respond to HTTP GET methods with content, see Section 19.10. Should return SOAP_OK, or a gSOAP error code. Built-in gSOAP function: http_get

int (*soap.fput)(**struct** soap *soap)

Called by the main server loop upon an HTTP PUT request. The SOAP_PUT_METHOD error is returned by default. This callback can be used to respond to HTTP PUT. Should return SOAP_OK, or a gSOAP error code. Built-in gSOAP function: http_put

int (*soap.fdel)(**struct** soap *soap)

Called by the main server loop upon an HTTP DELETE request. The SOAP_DELETE_METHOD error is returned by default. This callback can be used to respond to HTTP DELETE methods. Should return SOAP_OK, or a gSOAP error code. Built-in gSOAP function: http_del

int (*soap.fhead)(**struct** soap *soap)

Called by the main server loop upon an HTTP HEAD request. The SOAP_HEAD_METHOD error is returned by default. This callback can be used to respond to HTTP HEAD methods. Should return SOAP_OK, or a gSOAP error code. Built-in gSOAP function: http_get

int (*soap.fform)(**struct** soap *soap)

Called by the main server loop when a user-defined fparsehdr callback returned SOAP_FORM to signal that the HTTP body must be processed by this form handler callback. The HTTP POST form data MUST be read, otherwise keep-alive messages will end up out of sync. Should return SOAP_OK or a gSOAP error code. Built-in gSOAP function: none.

int (*soap.fpost)(**struct** soap *soap, **const char** *endpoint, **const char** *host, **int** port, **const char** *path, **const char** *action, size_t count)

Called from a client proxy to generate the HTTP header to connect to endpoint. Input parameters host, port, and path are micro-parsed from endpoint, action is the SOAP action, and count is the length of the SOAP message or 0 when SOAP_ENC_XML is set or when SOAP_IO_LENGTH is reset. Use function soap_send(**struct** soap *soap, **char** *s) to write the header contents. Should return SOAP_OK, or a gSOAP error code. Built-in gSOAP function: http_post.

int (*soap.fposthdr)(**struct** soap *soap, **const char** *key, **const char** *val)

Called by http_post and http_response (through the callbacks). Emits HTTP key: val header entries. Should return SOAP_OK, or a gSOAP error code. Built-in gSOAP function: http_post_header.

int (*soap.fresponse)(**struct** soap *soap, **int** soap_error_code, size_t count)

Called from a service to generate the response HTTP header. Input parameter soap_error_code is a gSOAP error code (see Section 10.2 and count is the length of the SOAP message or 0 when SOAP_ENC_XML is set or when SOAP_IO_LENGTH is reset. Use function soap_send(**struct** soap *soap, **char** *s) to write the header contents. Should return SOAP_OK, or a gSOAP error code Built-in gSOAP function: http_response

int (*soap.fparse)(**struct** soap *soap)

Called by client proxy and service to parse an HTTP header (if present). When user-defined, this routine must at least skip the header. Use function **int** soap_getline(**struct** soap *soap, **char** *buf, **int** len) to read HTTP header lines into a buffer buf of length len (returns empty line at end of HTTP header). Should return SOAP_OK, or a gSOAP error code. Built-in gSOAP function: http_parse

int (*soap.fparsehdr)(**struct** soap *soap, **const char** *key, **const char** *val)

Called by http_parse (through the fparse callback). Handles HTTP key: val header entries to set gSOAP's internals. Should return SOAP_OK, SOAP_STOP (see fstop) or a gSOAP error code. Built-in gSOAP function: http_parse_header

Callback (function pointer)

int (*soap.fsend)(**struct** soap *soap, **const char** *s, size_t n)

Called for all send operations to emit contents of s of length n. Should return SOAP_OK, or a gSOAP error code. Built-in gSOAP function: fsend

size_t (*soap.frecv)(**struct** soap *soap, **char** *s, size_t n)

Called for all receive operations to fill buffer s of maximum length n. Should return the number of bytes read or 0 in case of an error, e.g. EOF. Built-in gSOAP function: frecv

int (*soap.fignore)(**struct** soap *soap, **const char** *tag)

Called when an unknown XML element was encountered on the input. The tag parameter is the offending XML element tag name. Should return SOAP_OK, or a gSOAP error code such as SOAP_TAG_MISMATCH to throw an exception. Built-in gSOAP function: none.

int (*soap.fconnect)(**struct** soap *soap, **const char** *endpoint, **const char** *host, **int** port)

When non-NULL, this callback is called for all client-to-server connect operations instead of the built-in socket connect code. Therefore, it can be used to override the built-in connection establishment. Parameter endpoint contains the server endpoint URL, host the domain name or IP, and port the port number. Should return SOAP_OK, or a gSOAP error code. Built-in gSOAP function: none

SOAP_SOCKET (*soap.faccept)(**struct** soap *soap, SOAP_SOCKERT s, **struct** sockaddr *a, **int** *n)

Called by soap_accept. This is a wrapper routine for accept. Given master socket s should return a valid socket descriptor or SOAP_INVALID_SOCKET and set soap->error to an error code. Built-in gSOAP function: tcp_accept

int (*soap.fresolve)(**struct** soap *soap, **const char** *addr, **struct** in_addr *inaddr)

Called by soap_bind if a host name is given and soap_connect to resolve a domain name addr. Should set in_addr *a and return SOAP_OK or return SOAP_ERR upon failure.

Built-in gSOAP function: tcp_gethost

int (*soap.fpoll)(**struct** soap *soap)

Used by clients to check if the server is still responsive.

Built-in gSOAP function: soap_poll

int (*soap.fserveloop)(**struct** soap *soap)

Called after successful invocation of a server operation in the server loop, immediately after sending the response to a client. Can be used to clean up resources (e.g. using soap_end()) while serving a long sequence of keep-alive connections. Should return SOAP_OK, or set soap->error to a gSOAP error code and return soap->error. Built-in gSOAP function: none.

void (*soap.fmalloc)(**struct** soap *soap, size_t n)

Use to override memory allocation for deserialized C data. Memory allocated via this callback will *not* be automatically released by the gSOAP engine. The application must release this data by keeping track of the allocations. Note: it is not safe to traverse deserialized data structures and free each node, since data might be shared (SOAP multiref) and some allocated data such as the HTTP SOAPAction might no be part of the structure.

Built-in gSOAP function: none.

int (*soap.fheader)(**struct** soap *soap)

Called immediately after parsing a SOAP Header. The SOAP Header struct referenced by soap->header can be inspected and verified. The function should return SOAP_OK or a fault.

Built-in gSOAP function: none.

void (*soap.fsvalidate)(**struct** soap *soap, **const char** *pattern, **const char** *string)

Called to validate a non-NULL string against a non-NULL pattern. Patterns use XML schema regex syntax. Allows user-defined pattern validation. Should return SOAP_OK when the string matches the pattern or SOAP_PATTERN when the string does not match.

Built-in gSOAP function: none.

void (*soap.fwvalidate)(**struct** soap *soap, **const char** *pattern, **const** wchar_t *string)

Called to validate a non-NULL wide string against a non-NULL pattern. Patterns use XML schema regex syntax. Allows user-defined pattern validation. Should return SOAP_OK when the string matches the pattern or SOAP_PATTERN when the string does not match.

Built-in gSOAP function: none.

233

void (*soap.fseterror)(**struct** soap *soap, **const char** **code, **const char** **string)

Called to set the SOAP Fault code and string values based on the value of soap->error. Allows user-defined messages to be associated with gSOAP error codes to override gSOAP's built-in error messages.

Built-in gSOAP function: none.

In addition, a **void***user field in the **struct** soap data structure is available to pass user-defined data to the callbacks.

The following example uses I/O function callbacks for customized serialization of data into a fixed-size buffer and deserialization back into a datastructure:

```
char buf[10000]; // XML buffer
int len1 = 0; // #chars written
int len2 = 0; // #chars read
// mysend: put XML in buf[]
int mysend(struct soap *soap, const char *s, size_t n)
{
  if (len1 + n > sizeof(buf))
    return SOAP_EOF;
  strcpy(buf + len1, s);
  len1 += n;
  return SOAP_OK;
}
// myrecv: get XML from buf[]
size_t myrecv(struct soap *soap, char *s, size_t n)
{
  if (len2 + n > len1)
    n = len1 - len2;
  strncpy(s, buf + len2, n);
  len2 += n;
  return n;
}
int main()
{
  struct soap soap;
  struct ns__person p;
  soap_init(&soap);
  len1 = len2 = 0; // reset buffer pointers
  p.name = "John Doe";
  p.age = 25;
  soap.fsend = mysend; // assign callback
  soap.frecv = myrecv; // assign callback
  soap_begin_send(&soap);
  soap_set_omode(&soap, SOAP_XML_TREE);
  soap_serialize_ns__person(&soap, &p);
  soap_put_ns__person(&soap, &p, "ns:person", NULL);
  soap_end_send(&soap);
  if (soap.error)
  {
    soap_print_fault(&soap, stdout);
    exit(1);
  }
  soap_begin_recv(&soap);
  soap_get_ns__person(&soap, &p, "ns:person", NULL);
  soap_end_recv(&soap);
  if (soap.error)
  {
```

```
      soap_print_fault(&soap, stdout);
      exit(1);
   }
   soap_destroy(&soap);
   soap_end(&soap);
   soap_done(&soap); // disable callbacks
}
```

A fixed-size buffer to store the outbound message sent is not flexible to handle large content. To store the message in an expanding buffer, use for example:

```
struct buffer
{
   size_t len;
   size_t max;
   char *buf;
};
int main()
{
   struct buffer *h = malloc(sizeof(struct buffer));
   h->len = 0;
   h->max = 0;
   h->buf = NULL;
   soap.user = (void *)h; // pass buffer as a handle to the callback
   soap.fsend = mysend; // assign callback

   ...
   if (h->len)
   {
      ... // use h->buf[0..h->len-1] content
      // then cleanup:
      h->len = 0;
      h->max = 0;
      free(h->buf);
      h->buf = NULL;
   }
   ...
}
int mysend(struct soap *soap, const char *s, size_t n)
{
   struct buffer *h = (struct buffer*)soap->user; // get buffer through handle
   int m = h->max, k = h->len + n;
   // need to increase space?
   if (m == 0)
      m = 1024;
   else
      while (k >= m)
         m *= 2;
   if (m != h->max)
   {
      char *buf = malloc(m);
      memcpy(buf, h->buf, h->len);
```

```
        h->max = m;
        if(h->buf)
            free(h->buf);
        h->buf = buf;
    }
    memcpy(h->buf + h->len, s, n);
    h->len += n;
    return SOAP_OK;
}
```

The soap_done function can be called to reset the callback to the default internal gSOAP I/O and HTTP handlers.

The following example illustrates customized I/O and (HTTP) header handling. The SOAP request is saved to a file. The client proxy then reads the file contents as the service response. To perform this trick, the service response has exactly the same structure as the request. This is declared by the **struct** ns_ _test output parameter part of the service operation declaration. This struct resembles the service request (see the generated soapStub.h file created from the header file).

The header file is:

```
//gsoap ns service name: callback
//gsoap ns service namespace: urn:callback
struct ns_ _person
{
    char *name;
    int age;
};
int ns_ _test(struct ns_ _person in, struct ns_ _test &out);
```

The client program is:

```
#include "soapH.h"
...
SOAP_SOCKET myopen(struct soap *soap, const char *endpoint, const char *host, int port)
{
    if (strncmp(endpoint, "file:", 5))
    {
        printf("File name expected\n");
        return SOAP_INVALID_SOCKET;
    }
    if ((soap->sendfd = soap->recvfd = open(host, O_RDWR|O_CREAT, S_IWUSR|S_IRUSR)) < 0)
        return SOAP_INVALID_SOCKET;
    return soap->sendfd;
}
void myclose(struct soap *soap)
{
    if (soap->sendfd > 2) // still open?
        close(soap->sendfd); // then close it
    soap->recvfd = 0; // set back to stdin
    soap->sendfd = 1; // set back to stdout
```

```
}
int mypost(struct soap *soap, const char *endpoint, const char *host, const char *path, const
char *action, size_t count)
{
  return soap_send(soap, "Custom-generated file\n"); // writes to soap->sendfd
}
int myparse(struct soap *soap)
{
  char buf[256];
  if (lseek(soap->recvfd, 0, SEEK_SET) < 0 || soap_getline(soap, buf, 256)) // go to begin and
skip custom header
    return SOAP_EOF;
  return SOAP_OK;
}
int main()
{
  struct soap soap;
  struct ns__test r;
  struct ns__person p;
  soap_init(&soap); // reset
  p.name = "John Doe";
  p.age = 99;
  soap.fopen = myopen; // use custom open
  soap.fpost = mypost; // use custom post
  soap.fparse = myparse; // use custom response parser
  soap.fclose = myclose; // use custom close
  soap_call_ns__test(&soap, "file://test.xml", "", p, r);
  if (soap.error)
  {
    soap_print_fault(&soap, stdout);
    exit(1);
  }
  soap_end(&soap);
  soap_init(&soap); // reset to default callbacks
}
```

SOAP 1.1 and 1.2 specify that XML elements may be ignored when present in a SOAP payload on the receiving side. gSOAP ignores XML elements that are unknown, unless the XML attribute `mustUnderstand="true"` is present in the XML element. It may be undesirable for elements to be ignored when the outcome of the omission is uncertain. The soap.fignore callback can be set to a function that returns SOAP_OK in case the element can be safely ignored, or SOAP_MUSTUNDERSTAND to throw an exception, or to perform some application-specific action. For example, to throw an exception as soon as an unknown element is encountered on the input, use:

```
int myignore(struct soap *soap, const char *tag)
{
  return SOAP_MUSTUNDERSTAND; // never skip elements (secure)
}
...
soap.fignore = myignore;
soap_call_ns__method(&soap, ...); // or soap_serve(&soap);
```

To selectively throw an exception as soon as an unknown element is encountered but element `ns:xyz` can be safely ignored, use:

```
int myignore(struct soap *soap, const char *tag)
{
   if (soap_match_tag(soap, tag, "ns:xyz") != SOAP_OK)
      return SOAP_MUSTUNDERSTAND;
   return SOAP_OK;
}
...
soap.fignore = myignore;
soap_call_ns__method(&soap, ...); // or soap_serve(&soap)
...
struct Namespace namespaces[] =
{
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
   {"SOAP-ENC","http://schemas.xmlsoap.org/soap/encoding/"},
   {"xsi", "http://www.w3.org/2001/XMLSchema-instance"},
   {"xsd", "http://www.w3.org/2001/XMLSchema"},
   {"ns", "some-URI"}, // the namespace of element ns:xyz
   {NULL, NULL}
```

Function `soap_match_tag` compares two tags. The third parameter may be a pattern where * is a wildcard and - is a single character wildcard. So for example `soap_match_tag(tag, "ns:*")` will match any element in namespace `ns` or when no namespace prefix is present in the XML message.

The callback can also be used to keep track of unknown elements in an internal data structure such as a list:

```
struct Unknown
{
   char *tag;
   struct Unknown *next;
};
int myignore(struct soap *soap, const char *tag)
{
   char *s = (char*)soap_malloc(soap, strlen(tag)+1);
   struct Unknown *u = (struct Unknown*)soap_malloc(soap, sizeof(struct Unknown));
   if (s && u)
   {
      strcpy(s, tag);
      u->tag = s;
      u->next = ulist;
      ulist = u;
   }
}
...
struct soap *soap;
struct Unknown *ulist = NULL;
soap_init(&soap);
```

```
soap.fignore = myignore;
soap_call_ns_ _method(&soap, ...); // or soap_serve(&soap)
// print the list of unknown elements
soap_end(&soap); // clean up
```

19.8   HTTP 1.0 and 1.1

gSOAP uses HTTP 1.1 by default. You can revert to HTTP 1.0 as follows:

```
struct soap soap;
soap_init(&soap);
...
soap.http_version = "1.0";
```

This sets the HTTP version and reconfigures the engine to revert to HTTP 1.0. Note that you cannot use HTTP chunking with HTTP 1.0.

19.9   HTTP 307 Temporary Redirect Support

The client-side handling of HTTP 307 code "Temporary Redirect" and any of the redirect codes 301, 302, and 303 are not automated in gSOAP. Client application developers may want to consider adding a few lines of code to support redirects. It was decided not to automatically support redirects for the following reasons:

- Redirecting a secure HTTPS address to a non-secure HTTP address via 307 creates a security vulnerability.

- Cyclic redirects must be detected (e.g. allowing only a limited number of redirect levels).

- Redirecting HTTP POST will result in re-serialization and re-post of the entire SOAP request. The SOAP request message must be re-posted in its entirety when re-issuing the SOAP operation to a new address.

To implement client-side 307 redirect, add the following lines of code:

```
char *endpoint = NULL; // use default endpoint given in WSDL (or add another one here)
int n = 10; // max redirect count
while (n− −)
{
  if (soap_call_ns1_ _myMethod(soap, endpoint, ...))
  {
    if ((soap−>error >= 301 && soap−>error <= 303) || soap−>error == 307)
      endpoint = soap_strdup(soap, soap−>endpoint); // endpoint from HTTP 301, 302, 303, 307
Location header
    else
      { ... report and handle error
        break;
      }
```

```
      }
   else
      break;
}
```

19.10   HTTP GET Support

To implement your own HTTP (HTTPS) GET request responses, you need to set the soap.fget
callback. The callback is required to produce a response to the request in textual form, such as a
Web page or a SOAP/XML response. This method does not work with CGI.

The following example produces a Web page upon a HTTP GET request (e.g. from a browser):

```
struct soap *soap = soap_new();
soap->fget = http_get;
...
soap_serve(soap);
...
int http_get(struct soap *soap)
{
   soap_response(soap, SOAP_HTML); // HTTP response header with text/html
   soap_send(soap, "¡HTML¿My Web server is operational.¡/HTML¿");
   soap_end_send(soap);
   return SOAP_OK;
}
```

The example below produces a WSDL file upon a HTTP GET with path ?wsdl:

```
int http_get(struct soap *soap)
{
   FILE *fd = NULL;
   char *s = strchr(soap->path, '?');
   if (!s || strcmp(s, "?wsdl"))
      return SOAP_GET_METHOD;
   fd = fopen("myservice.wsdl", "rb"); // open WSDL file to copy
   if (!fd)
      return 404; // return HTTP not found error
   soap->http_content = "text/xml"; // HTTP header with text/xml content
   soap_response(soap, SOAP_FILE);
   for (;;)
   {
      size_t r = fread(soap->tmpbuf, 1, sizeof(soap->tmpbuf), fd);
      if (!r)
         break;
      if (soap_send_raw(soap, soap->tmpbuf, r))
         break; // can't send, but little we can do about that
   }
   fclose(fd);
   soap_end_send(soap);
   return SOAP_OK;
}
```

Using one-way SOAP/XML message, you can also return a SOAP/XML response:

```
int http_get(struct soap *soap)
{
    if ((soap->omode & SOAP_IO) != SOAP_IO_CHUNK)
        soap_set_omode(soap, SOAP_IO_STORE); // if not chunking we MUST buffer entire content
to determine content length
    soap_response(soap, SOAP_OK);
    return soap_send_ns1__mySendMethodResponse(soap, "", NULL, ... params ...);
}
```

where ns1__mySendMethodResponse is a one-way message declared in a gSOAP header file as:

```
int ns1__mySendMethodResponse(... params ..., void);
```

The generated soapClient.cpp includes the sending-side stub function.

19.11   TCP and HTTP Keep-Alive

gSOAP supports keep-alive socket connections. To activate keep-alive support, set the SOAP_IO_KEEPALIVE flag for both input and output modes, see Section 9.12. For example

```
struct soap soap;
soap_init2(&soap, SOAP_IO_KEEPALIVE, SOAP_IO_KEEPALIVE);
```

When a client or a service communicates with another client or service that supports keep alive, the attribute soap.keep_alive will be set to 1, otherwise it is reset to 0 (indicating that the other party will close the connection). The connection maybe terminated on either end before the communication completed, for example when the server keep-alive connection has timed out. This generates a "Broken Pipe" signal on Unix/Linux platforms. This signal can be caught with a signal handler:

```
signal(SIGPIPE, sigpipe_handle);
```

where, for example:

```
void sigpipe_handle(int x) { }
```

Alternatively, broken pipes can be kept silent by setting:

```
soap.socket_flags = MSG_NOSIGNAL;
```

This setting will not generate a sigpipe but read/write operations return SOAP_EOF instead. Note that Win32 systems do not support signals and lack the MSG_NOSIGNAL flag. The sigpipe handling and flags are not very portable.

A connection will be kept open only if the request contains an HTTP 1.0 header with "Connection: Keep-Alive" or an HTTP 1.1 header that does not contain "Connection: close". This means

that a gSOAP client method call should use "`http://`" in the endpoint URL of the request to the stand-alone service to ensure HTTP headers are used.

If the client does not close the connection, the server will wait forever when no recv_timeout is specified. In addition, other clients will be denied service as long as a client keeps the connection to the server open. To prevent this from happening, the service should be multi-threaded such that each thread handles the client connection:

```
int main(int argc, char **argv)
{
   struct soap soap, *tsoap;
   pthread_t tid;
   int m, s;
   soap_init2(&soap, SOAP_IO_KEEPALIVE, SOAP_IO_KEEPALIVE);
   soap.max_keep_alive = 100; // at most 100 calls per keep-alive session
   soap.accept_timeout = 600; // optional: let server time out after ten minutes of inactivity
   m = soap_bind(&soap, NULL, 18000, BACKLOG); // use port 18000 on the current machine
   if (m < 0)
   {
      soap_print_fault(&soap, stderr);
      exit(1);
   }
   fprintf(stderr, "Socket connection successful %d\n", m);
   for (count = 0; count >= 0; count++)
   {
      soap.socket_flags = MSG_NOSIGNAL; // use this
      soap.accept_flags = SO_NOSIGPIPE; // or this to prevent sigpipe
      s = soap_accept(&soap);
      if (s < 0)
      {
         if (soap.errnum)
            soap_print_fault(&soap, stderr);
         else
            fprintf(stderr, "Server timed out\n"); // Assume timeout is long enough for threads to
complete serving requests
         break;
      }
      fprintf(stderr, "Accepts socket %d connection from IP %d.%d.%d.%d\n", s, (int)(soap.ip>>24)&0xFF,
(int)(soap.ip>>16)&0xFF, (int)(soap.ip>>8)&0xFF, (int)soap.ip&0xFF);
      tsoap = soap_copy(&soap);
      pthread_create(&tid, NULL, (void*(*)(void*))process_request, (void*)tsoap);
   }
   return 0;
}
void *process_request(void *soap)
{
   pthread_detach(pthread_self());
   ((struct soap*)soap)->recv_timeout = 300; // Timeout after 5 minutes stall on recv
   ((struct soap*)soap)->send_timeout = 60; // Timeout after 1 minute stall on send
   soap_serve((struct soap*)soap);
   soap_destroy((struct soap*)soap);
   soap_end((struct soap*)soap);
```

```
        soap_free((struct soap*)soap);
        return NULL;
    }
```

To prevent a malicious client from keeping a thread waiting forever by keeping the connection open, timeouts are set in the process_request routine as shown. See Section 19.19 for more details on timeout settings.

A gSOAP client call will automatically attempt to re-establish a connection to a server when the server has terminated the connection for any reason. This way, a sequence of calls can be made to the server while keeping the connection open. Client stubs will poll the server to check if the connection is still open. When the connection was terminated by the server, the client will automatically reconnect.

A client should reset SOAP_IO_KEEPALIVE just before the last call to a server to close the connection after this last call. This will close the socket after the call and also informs the server to gracefully close the connection.

The client-side can also set the TCP keep-alive socket properties, using the soap.tcp_keep_alive flag (set to 1 to enable), soap.tcp_keep_idle to set the TCP_KEEPIDLE value, soap.tcp_keep_intvl to set the TCP_KEEPINTVL value, and soap.tcp_keep_cnt to set the TCP_KEEPCNT value.

If a client is in the middle of soap call that might take a long time and the server goes away/down the caller does not get any feedback until the soap.recv_timeout is reached. Enabling TCP keep alive on systems that support it allows for a faster connection teardown detection for applications that need it.

19.12   HTTP Chunked Transfer Encoding

gSOAP supports HTTP chunked transfer encoding. Un-chunking of inbound messages takes place automatically. Outbound messages are never chunked, except when the SOAP_IO_CHUNK flag is set for the output mode. Most Web services, however, will not accept chunked inbound messages.

19.13   HTTP Buffered Sends

The entire outbound message can be stored to determine the HTTP content length rather than the two-phase encoding used by gSOAP which requires a separate pass over the data to determine the length of the outbound message. Setting the flag SOAP_IO_STORE for the output mode will buffer the entire message. This can speed up the transmission of messages, depending on the content, but may require significant storage space to hold the verbose XML message.

Zlib compressed transfers require buffering. The SOAP_IO_STORE flag is set when the SOAP_ENC_ZLIB flag is set to send compressed messages. The use of chunking significantly reduces memory usage and may speed up the transmission of compressed SOAP/XML messages. This is accomplished by setting the SOAP_IO_CHUNK flag with SOAP_ENC_ZLIB for the output mode.

19.14   HTTP Authentication

HTTP authentication (basic) is enabled at the client-side by setting the soap.userid and soap.passwd strings to a username and password, respectively. A server may request user authentication and denies access (HTTP 401 error) when the client tries to connect without HTTP authentication (or with the wrong authentication information).

Here is an example client code fragment to set the HTTP authentication username and password:

```
struct soap soap;
soap_init(&soap);
soap.userid = "guest";
soap.passwd = "visit";
...
```

A client SOAP request will have the following HTTP header:

```
POST /XXX HTTP/1.0
Host:  YYY
User-Agent:  gSOAP/2.2
Content-Type:  text/xml; charset=utf-8
Content-Length:  nnn
Authorization:  Basic Z3Vlc3Q6Z3Vlc3Q=
...
```

A client MUST set the soap.userid and soap.passwd strings for each call that requires client authentication. The strings are reset after each successful or unsuccessful call.

When present, the value of the WWW-Authenticate HTTP header with the authentication realm can be obtained from the soap.authrealm string. This is useful for clients to respond intelligently to authentication requests.

A stand-alone gSOAP Web Service can enforce HTTP authentication upon clients, by checking the soap.userid and soap.passwd strings. These strings are set when a client request contains HTTP authentication headers. The strings SHOULD be checked in each service method (that requires authentication to execute).

Here is an example service method implementation that enforced client authentication:

```
int ns__method(struct soap *soap, ...)
{
  if (!soap->.userid || !soap->.passwd || strcmp(soap->.userid, "guest") || strcmp(soap->.passwd,
"visit"))
      return 401;
...
}
```

When the authentication fails, the service response with a SOAP Fault message and a HTTP error code "401 Unauthorized". The HTTP error codes are described in Section 10.2.

19.15 HTTP NTLM Authentication

HTTP NTLM authentication is enabled at the client-side by installing libntlm from `http://www.nongnu.org/libntlm` and compiling all project source codes with -DWITH_NTLM.

In your application code set the soap.userid, soap.passwd, and soap.authrealm strings to a username, password, and the authentication domain respectively. A server may request NTLM authentication and denies access (HTTP 401 authentication required or HTTP 407 HTTP proxy authentication required) when the client tries to connect without HTTP authentication (or with the wrong authentication information).

Here is an example client code fragment to set the NTLM authentication username and password:

```
struct soap soap;
soap_init1(&soap, SOAP_IO_KEEPALIVE);
if (soap_call_ns__method(&soap, ...))
{ if (soap.error == 401)
  { soap.userid = "Zaphod";
    soap.passwd = "Beeblebrox";
    soap.authrealm = "Ursa-Minor";
    if (soap_call_ns__method(&soap, ...))
      ...
```

The following NTLM handshake between the client C and server S is performed:

```
1:  C --> S  POST ...
             Content-Type:  text/xml; charset=utf-8

2:  C <-- S  401 Unauthorized
             WWW-Authenticate:  NTLM

3:  C --> S  GET ...
             Authorization:  NTLM <base64-encoded type-1-message>

4:  C <-- S  401 Unauthorized
             WWW-Authenticate:  NTLM <base64-encoded type-2-message>

5:  C --> S  POST ...
             Content-Type:  text/xml; charset=utf-8
             Authorization:  NTLM <base64-encoded type-3-message>

6:  C <-- S  200 OK
```

where stages 1 and 2 indicates a client attempting to connect without authorization information, which is the first method call in the code above. Stage 3 to 6 happen with the proper client authentication set with soap.userid, soap.passwd, and soap.authrealm provided. NTLM authenticates connections, not requests. When the connection is kept alive, subsequent messages can be exchanged without re-authentication.

To avoid the overhead of the first rejected call, use:

```
struct soap soap;
soap_init1(&soap, SOAP_IO_KEEPALIVE);
```

```
soap.userid = "Zaphod";
soap.passwd = "Beeblebrox";
soap.authrealm = "Ursa-Minor";
soap.ntlm_challenge = "";
if (soap_call_ns__method(&soap, ...))
   ...
```

When the authentication fails (stage 1 and 2), the service response with a SOAP Fault message and a HTTP error code "401 Unauthorized". The HTTP error codes are described in Section 10.2.

On windows, an alternative is to use the WinInet module, which has built-in NTLM support. The WinInet for gSOAP module is available in the mod_gsoap directory of the gSOAP package. Instructions for WinInet use are included there.

19.16 HTTP Proxy NTLM Authentication

For HTTP 407 Proxy Authentication Required, set the proxy userid and passwd:

```
struct soap soap;
soap_init1(&soap, SOAP_IO_KEEPALIVE);
soap.proxy_host = "...";
soap.proxy_port = ...;
if (soap_call_ns__method(&soap, ...))
{ if (soap.error == 407)
   { soap.proxy_userid = "Zaphod";
     soap.proxy_passwd = "Beeblebrox";
     soap.authrealm = "Ursa-Minor";
     if (soap_call_ns__method(&soap, ...))
        ...
```

To avoid the overhead of the first rejected call, use:

```
struct soap soap;
soap_init1(&soap, SOAP_IO_KEEPALIVE);
soap.proxy_host = "...";
soap.proxy_port = ...;
soap.proxy_userid = "Zaphod";
soap.proxy_passwd = "Beeblebrox";
soap.authrealm = "Ursa-Minor";
soap.ntlm_challenge = "";
if (soap_call_ns__method(&soap, ...))
    ...
```

19.17 HTTP Proxy Basic Authentication

HTTP proxy authentication (basic) is enabled at the client-side by setting the soap.proxy_userid and soap.proxy_passwd strings to a username and password, respectively. For example, a proxy server may request user authentication. Otherwise, access is denied by the proxy (HTTP 407 error). Example client code fragment to set proxy server, username, and password:

```
struct soap soap;
soap_init(&soap);
soap.proxy_host = "xx.xx.xx.xx"; // IP or domain
soap.proxy_port = 8080;
soap.proxy_userid = "guest";
soap.proxy_passwd = "guest";
...
```

A client SOAP request will have the following HTTP header:

```
POST /XXX HTTP/1.0
Host:  YYY
User-Agent:  gSOAP/2.2
Content-Type:  text/xml; charset=utf-8
Content-Length:  nnn
Proxy-Authorization:  Basic Z3Vlc3Q6Z3Vlc3Q=
...
```

When X-Forwarded-For headers are returned by the proxy, the header can be accessed in the soap.proxy_from string.

The CONNECT method is used for HTTP proxy authentication:

```
CONNECT server.example.com:80 HTTP/1.1
```

In some cases, it may be necessary to use the Host HTTP header with the CONNECT protocol:

```
CONNECT server.example.com:80 HTTP/1.1
Host:  server.example.com:80
```

If so, compile the gSOAP code with -DWITH_CONNECT_HOST to include the Host HTTP header with the CONNECT protocol.

19.18   Messaging Speed and Performance Improvement Tips

Here are some tips you can use to speed up gSOAP. gSOAP's default settings are choosen to maximize portability and compatibility. The settings can be tweaked to optimize the performance as follows:

- Increase the buffer size SOAP_BUFLEN by changing the SOAP_BUFLEN macro in stdsoap2.h. Use buffer size $2^18 = 262144$ for example.

- Use HTTP keep-alive at the client-side, see 19.11, when the client needs to make a series of calls to the same server. Server-side keep-alive support can greatly improve performance of both client and server. But be aware that clients and services under Unix/Linux require signal handlers to catch dropped connections.

- Use HTTP chunked transfers, see 19.12.

- Do NOT use gzip compression, since the overhead of compression is typically higher than the bandwidth gains.

- Set the SOAP_XML_TREE flag to disable id-ref multi-ref object (de)serialization. This boosts performance significantly and works with SOAP document/literal style (i.e. no id-ref graph serialization as required with SOAP encoding style).

- Compile stdsoap2.c and stdsoap2.cpp with -DWITH_NOIDREF to improve performance even better by permanently disabling id-ref multi-ref object (de)serialization.

- Do NOT use DEBUG mode, since the overhead of logging is significant.

19.19 Timeout Management for Non-Blocking Operations

Socket connect, accept, send, and receive timeout values can be set to manage socket communication timeouts. The soap.connect_timeout, soap.accept_timeout, soap.send_timeout, and soap.recv_timeout attributes of the current gSOAP runtime context soap can be set to the appropriate user-defined socket send, receive, and accept timeout values. A positive value measures the timeout in seconds. A negative timeout value measures the timeout in microseconds (10^{-6} sec).

The soap.connect_timeout specifies the timeout for soap_call_ns__method calls.

The soap.accept_timeout specifies the timeout for soap_accept(&soap) calls.

The soap.send_timeout and soap.recv_timeout specify the timeout for non-blocking socket I/O operations.

Example:

```
struct soap soap;
soap_init(&soap);
soap.send_timeout = 10;
soap.recv_timeout = 10;
```

This will result in a timeout if no data can be send in 10 seconds and no data is received within 10 seconds after initiating a send or receive operation over the socket. A value of zero disables timeout, for example:

```
soap.send_timeout = 0;
soap.recv_timeout = 0;
```

When a timeout occurs in send/receive operations, a SOAP_EOF exception will be raised ("end of file or no input"). Negative timeout values measure timeouts in microseconds, for example:

```
#define uSec *-1
#define mSec *-1000
soap.accept_timeout = 10 uSec;
soap.send_timeout = 20 mSec;
soap.recv_timeout = 20 mSec;
```

The macros improve readability.

Caution: many Linux versions do not support non-blocking connect(). Therefore, setting soap.connect_timeout for non-blocking soap_call_ns_ _method calls may not work under Linux.

Caution: interrupts (EINTR) can affect the blocking time in I/O operations. The maximum number of EINTR that will not trigger an error is set by SOAP_MAXEINTR in stdsoap2.h, which is 10 by default. Each EINTR may increase the blocking time by up to one second, up to SOAP_MAXEINTR seconds total.

19.20   Socket Options and Flags

gSOAP's socket communications can be controlled with socket options and flags. The gSOAP run-time context **struct** soap flags are: **int** soap.socket_flags to control socket send() and recv() calls, **int** soap.connect_flags to set client connection socket options, **int** soap.bind_flags to set server-side port bind socket options, **int** soap.accept_flags to set server-side request message accept socket options. See the manual pages of send and recv for soap.socket_flags values and see the manual pages of setsockopt for soap.connect_flags, soap.bind_flags, and soap.accept_flags (SOL_SOCKET) values. These SO_ socket option flags (see setsockopt manual pages) can be bit-wise or-ed to set multiple socket options at once. The client-side flag soap.connect_flags=SO_LINGER is supported with values l_onoff=1 and l_linger=soap.linger_time. The soap.linger_time determines the wait time (the time resolution is system dependent, though according to some experts only zero and nonzero values matter). The linger option can be used to manage the number of connections that remain in TIME_WAIT state at the server side.

For example, to disable sigpipe signals on Unix/Linux platforms use: soap.socket_flags = MSG_NOSIGNAL and/or soap.connect_flags = SO_NOSIGPIPE (i.e. client-side connect) depending on your platform.

Use soap.bind_flags=SO_REUSEADDR to enable server-side port reuse and local port sharing (but be aware of the possible security implications such as port hijacking).

Note that multiple socket options can be explicitly set with setsockopt as follows:

```
int sock = soap_bind(soap, host, port, backlog);
if (soap_valid_socket(sock))
{
    setsockopt(sock, ..., ..., ..., ...);    setsockopt(sock, ..., ..., ..., ...);
```

19.21   Secure Web Services with HTTPS/SSL

When a Web Service is installed as CGI, it uses standard I/O that is encrypted/decrypted by the Web server that runs the CGI application. HTTPS/SSL support must be configured for the Web server (not CGI-based Web Service application itself).

To enable SSL for stand-alone gSOAP servers, first install OpenSSL and use option -DWITH_OPENSSL to compile the sources with your C or C++ compiler (or use -DWITH_GNUTLS if you prefer GNUTLS), for example:

```
> c++ -DWITH_OPENSSL -o myprog myprog.cpp stdsoap2.cpp soapC.cpp soapServer.cpp -lssl
-lcrypto
```

SSL support for stand-alone gSOAP Web services is enabled by calling soap_ssl_accept to perform
the SSL/TLS handshake after soap_accept. In addition, a key file, a CA file (or path to certificates),
DH file (if RSA is not used), and password need to be supplied. Instructions on how to do this can
be found in the OpenSSL documentation http://www.openssl.org. See also Section 19.24.

Let's take a look at an example SSL secure multi-threaded stand-alone SOAP Web Service:

```cpp
int main()
{
  int m, s;
  pthread_t tid;
  struct soap soap, *tsoap;
  soap_ssl_init(); /* init OpenSSL (just once) */
  if (CRYPTO_thread_setup()) // OpenSSL
  {
    fprintf(stderr, "Cannot setup thread mutex\n");
    exit(1);
  }
  soap_init(&soap);
  if (soap_ssl_server_context(&soap,
    SOAP_SSL_DEFAULT,
    "server.pem", /* keyfile: required when server must authenticate to clients (see SSL docs on
how to obtain this file) */
    "password", /* password to read the key file (not used with GNUTLS) */
    "cacert.pem", /* optional cacert file to store trusted certificates */
    NULL, /* optional capath to directory with trusted certificates */
    "dh512.pem", /* DH file name or DH key len bits (minimum is 512, e.g. "512") to generate
DH param, if NULL use RSA */
    NULL, /* if randfile!=NULL: use a file with random data to seed randomness */
    NULL /* optional server identification to enable SSL session cache (must be a unique name)
*/   ))
  {
    soap_print_fault(&soap, stderr);
    exit(1);
  }
  m = soap_bind(&soap, NULL, 18000, 100); // use port 18000
  if (m < 0)
  {
    soap_print_fault(&soap, stderr);
    exit(1);
  }
  fprintf(stderr, "Socket connection successful: master socket = %d\n", m);
  for (;;)
  {
    s = soap_accept(&soap);
    fprintf(stderr, "Socket connection successful: slave socket = %d\n", s);
    if (s < 0)
    {
      soap_print_fault(&soap, stderr);
```

```
            break;
        }
        tsoap = soap_copy(&soap); /* should call soap_ssl_accept on a copy */
        if (!tsoap)
            break;
        pthread_create(&tid, NULL, &process_request, (void*)tsoap);
    }
    soap_done(&soap); /* deallocates SSL context */
    CRYPTO_thread_cleanup(); // OpenSSL
    return 0;
}
void *process_request(void *soap)
{
    pthread_detach(pthread_self());
    if (soap_ssl_accept((struct soap*)soap))
        soap_print_fault(tsoap, stderr);
    else
        soap_serve((struct soap*)soap);
    soap_destroy((struct soap*)soap);
    soap_end((struct soap*)soap);
    soap_free((struct soap*)soap); // done and free context
    return NULL;
}
```

The soap_ssl_server_context function initializes the server-side SSL context. The server.pem key file is the server's private key concatenated with its certificate. The cacert.pem is used to authenticate clients and contains the client certificates. Alternatively a directory name can be specified. This directory is assumed to contain the certificates. The dh512.pem file specifies that DH will be used for key agreement instead of RSA. A numeric value greater than 512 can be provided instead as a string constant (e.g. "512") to allow the engine to generate the DH parameters on the fly (this can take a while) rather than retrieving them from a file. The randfile entry can be used to seed the PRNG. The last entry enable server-side session caching. A unique server name is required.

The GNUTLS mutex lock setup is automatically peformed in the gSOAP engine, but only when POSIX threads are detected and available.

OpenSSL requires mutex locks to be explicitly setup in your code for multithreaded applications, for which we need to call CRYPTO_thread_setup() and CRYPTO_thread_cleanup(). These routines can be found in openssl/crypto/threads/th-lock.c and are also used in the SSL example codes samples/ssl. These routines are required to setup locks for multi-threaded applications that use SSL.

We give a Windows and POSIX threads implementation of these here:

```
#include <unistd.h> /* defines _POSIX_THREADS if pthreads are available */
#ifdef _POSIX_THREADS
# include <pthread.h>
#endif
#if defined(WIN32)
# define MUTEX_TYPE HANDLE
# define MUTEX_SETUP(x) (x) = CreateMutex(NULL, FALSE, NULL)
# define MUTEX_CLEANUP(x) CloseHandle(x)
```

```c
# define MUTEX_LOCK(x) WaitForSingleObject((x), INFINITE)
# define MUTEX_UNLOCK(x) ReleaseMutex(x)
# define THREAD_ID GetCurrentThreadID()
#elif defined(_POSIX_THREADS)
# define MUTEX_TYPE pthread_mutex_t
# define MUTEX_SETUP(x) pthread_mutex_init(&(x), NULL)
# define MUTEX_CLEANUP(x) pthread_mutex_destroy(&(x))
# define MUTEX_LOCK(x) pthread_mutex_lock(&(x))
# define MUTEX_UNLOCK(x) pthread_mutex_unlock(&(x))
# define THREAD_ID pthread_self()
#else
# error "You must define mutex operations appropriate for your platform"
# error "See OpenSSL /threads/th-lock.c on how to implement mutex on your platform"
#endif
struct CRYPTO_dynlock_value { MUTEX_TYPE mutex; };
static MUTEX_TYPE *mutex_buf;
static struct CRYPTO_dynlock_value *dyn_create_function(const char *file, int line)
{
  struct CRYPTO_dynlock_value *value;
  value = (struct CRYPTO_dynlock_value*)malloc(sizeof(struct CRYPTO_dynlock_value));
  if (value)
    MUTEX_SETUP(value->mutex);
  return value;
}
static void dyn_lock_function(int mode, struct CRYPTO_dynlock_value *l, const char *file, int line)
{
  if (mode & CRYPTO_LOCK)
    MUTEX_LOCK(l->mutex);
  else
    MUTEX_UNLOCK(l->mutex);
}
static void dyn_destroy_function(struct CRYPTO_dynlock_value *l, const char *file, int line)
{
  MUTEX_CLEANUP(l->mutex);
  free(l);
}
void locking_function(int mode, int n, const char *file, int line)
{
  if (mode & CRYPTO_LOCK)
    MUTEX_LOCK(mutex_buf[n]);
  else
    MUTEX_UNLOCK(mutex_buf[n]);
}
unsigned long id_function()
{
  return (unsigned long)THREAD_ID;
}
int CRYPTO_thread_setup()
{
  int i;
```

```
      mutex_buf = (MUTEX_TYPE*)malloc(CRYPTO_num_locks() * sizeof(MUTEX_TYPE));
      if (!mutex_buf)
        return SOAP_EOM;
      for (i = 0; i < CRYPTO_num_locks(); i++)
        MUTEX_SETUP(mutex_buf[i]);
      CRYPTO_set_id_callback(id_function);
      CRYPTO_set_locking_callback(locking_function);
      CRYPTO_set_dynlock_create_callback(dyn_create_function);
      CRYPTO_set_dynlock_lock_callback(dyn_lock_function);
      CRYPTO_set_dynlock_destroy_callback(dyn_destroy_function);
      return SOAP_OK;
    }
    void CRYPTO_thread_cleanup()
    {
      int i;
      if (!mutex_buf)
        return;
      CRYPTO_set_id_callback(NULL);
      CRYPTO_set_locking_callback(NULL);
      CRYPTO_set_dynlock_create_callback(NULL);
      CRYPTO_set_dynlock_lock_callback(NULL);
      CRYPTO_set_dynlock_destroy_callback(NULL);
      for (i = 0; i < CRYPTO_num_locks(); i++)
        MUTEX_CLEANUP(mutex_buf[i]);
      free(mutex_buf);
      mutex_buf = NULL;
    }
```

For Unix and Linux, make sure you have signal handlers set in your service and/or client applications to catch broken connections (SIGPIPE):

```
    signal(SIGPIPE, sigpipe_handle);
```

where, for example:

```
    void sigpipe_handle(int x) { }
```

By default, clients are not required to authenticate. To support client authentication use the following:

```
    if (soap_ssl_server_context(&soap,
      SOAP_SSL_REQUIRE_CLIENT_AUTHENTICATION,
      "server.pem",
      "password",
      "cacert.pem",
      NULL,
      "dh512.pem",
      NULL,
      NULL
    ))
```

```
      {
        soap_print_fault(&soap, stderr);
        exit(1);
      }
```

This requires each client to authenticate with its certificate.

Since release version 2.8.20 SSL v3 is disabled. To enable SSL v3 together with TLS v1 use SOAP_SSLv3_TLSv1 in soap_ssl_server_context.

The cacert file and capath are optional. Either one can be specified when clients must run on non-trusted systems (capath is not used with GNUTLS). We want to avoid storing trusted certificates in the default location on the file system when that is not secure. Therefore, a flat cacert.pem file or directory can be specified to store trusted certificates.

The gSOAP distribution includes a cacerts.pem file with the certificates of all certificate authorities such as Verisign. You can use this file to verify the authentication of servers that provide certificates issued by these CAs.

The cacert.pem, client.pem, and server.pem files in the gSOAP distribution are examples of self-signed certificates. The client.pem and server.pem contain the client/server private key concatenated with the certificate. The keyfiles (client.pem and server.pem) are created by concatenating the private key PEM with the certificate PEM. The keyfile SHOULD NEVER be shared with any party. With OpenSSL, you can encrypt the keyfiles with a password to offer some protection and the password is used in the client/server code to read the keyfile. GNUTLS does not support this feature and cannot encrypt or decrypt a keyfile.

Caution: it is important that the WITH_OPENSSL macro MUST be consistently defined to compile the sources, such as stdsoap2.cpp, soapC.cpp, soapClient.cpp, soapServer.cpp, and all application sources that include stdsoap2.h or soapH.h. If the macros are not consistently used, the application will crash due to a mismatches in the declaration and access of the gSOAP context.

19.22   Secure Clients with HTTPS/SSL

To utilize HTTPS/SSL, you need to install the OpenSSL library on your platform or GNUTLS for a light-weight SSL/TLS library. After installation, compile all the sources of your application with option -DWITH_OPENSSL (or -DWITH_GNUTLS when using GNUTLS). For example on Linux:

```
> c++ -DWITH_OPENSSL myclient.cpp stdsoap.cpp soapC.cpp soapClient.cpp -lssl -lcrypto
```

or Unix:

```
> c++ -DWITH_OPENSSL myclient.cpp stdsoap.cpp soapC.cpp soapClient.cpp -lxnet -lsocket -lnsl
-lssl -lcrypto
```

or you can add the following line to soapdefs.h:

```
#define WITH_OPENSSL
```

and compile with option -DWITH_SOAPDEFS_H to include soapdefs.h in your project. A client program simply uses the prefix https: instead of http: in the endpoint URL of a service operation call to a Web Service to use encrypted transfers (if the service supports HTTPS). You need to specify the client-side key file and password of the keyfile:

```
soap_ssl_init(); /* init OpenSSL (just once) */
if (soap_ssl_client_context(&soap,
  SOAP_SSL_DEFAULT,
  "client.pem", /* keyfile: required only when client must authenticate to server (see SSL docs on
how to obtain this file) */
  "password", /* password to read the key file (not used with GNUTLS) */
  "cacerts.pem", /* cacert file to store trusted certificates (needed to verify server) */   NULL, /*
capath to directory with trusted certificates */
  NULL /* if randfile!=NULL: use a file with random data to seed randomness */
))
{
  soap_print_fault(&soap, stderr);
  exit(1);
}
soap_call_ns__mymethod(&soap, "https://domain/path/secure.cgi", "", ...);
```

By default, server authentication is enabled and the cacerts.pem or capath (not used with GNUTLS) must be set so that the CA certificates of the server(s) are accessible at run time. The cacerts.pem file included in the package contains the certificates of common CAs. This file must be supplied with the client, if server authentication is required. Althernatively, you can use the plugin/cacerts.h and plugin/cacerts.c code to embed CA certificates in your client code.

Other client-side SSL options are SOAP_SSL_SKIP_HOST_CHECK to skip the host name verification check and SOAP_SSL_ALLOW_EXPIRED_CERTIFICATE to allow connecting to a host with an expired certificate. For example,

```
soap_ssl_init(); /* init OpenSSL (just once) */
if (soap_ssl_client_context(&soap,
  SOAP_SSL_REQUIRE_SERVER_AUTHENTICATION
  — SOAP_SSL_SKIP_HOST_CHECK,
  — SOAP_SSL_ALLOW_EXPIRED_CERTIFICATE,
  "client.pem", /* keyfile: required only when client must authenticate to server (see SSL docs on
how to obtain this file) */
  "password", /* password to read the key file (not used with GNUTLS) */
  "cacerts.pem", /* cacert file to store trusted certificates (needed to verify server) */   NULL, /*
capath to directory with trusted certificates */
  NULL /* if randfile!=NULL: use a file with random data to seed randomness */
))
{
  soap_print_fault(&soap, stderr);
  exit(1);
}
soap_call_ns__mymethod(&soap, "https://domain/path/secure.cgi", "", ...);
```

For systems based on Microsoft windows, the WinInet module can be used instead, see mod_gsoap/gsoap_win/wininet.

Since release version 2.8.20 SSL v3 is disabled. To enable SSL v3 together with TLS v1 use SOAP_SSLv3_TLSv1 in soap_ssl_client_context.

To disable server authentication for testing purposes, use the following:

```
if (soap_ssl_client_context(&soap,
    SOAP_SSL_NO_AUTHENTICATION,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
))
{
    soap_print_fault(&soap, stderr);
    exit(1);
}
```

This also assumes that the server does not require clients to authenticate (the keyfile is absent).

Make sure you have signal handlers set in your application to catch broken connections (SIGPIPE):

```
signal(SIGPIPE, sigpipe_handle);
```

where, for example:

```
void sigpipe_handle(int x) { }
```

Caution: it is important that the WITH_OPENSSL macro MUST be consistently defined to compile the sources, such as stdsoap2.cpp, soapC.cpp, soapClient.cpp, soapServer.cpp, and all application sources that include stdsoap2.h or soapH.h. If the macros are not consistently used, the application will crash due to a mismatches in the declaration and access of the gSOAP context. **Caution**: concurrent client calls MUST be made using separate soap structs copied with soap_copy from an originating struct initialized with soap_ssl_client_context. In addition, the thread initialization code discussed in Section 19.21 MUST be used to properly setup OpenSSL in a multi-threaded client application.

19.23   SSL Authentication Callbacks

The fsslauth callback function controls OpenSSL/GNUTLS authentication initialization:

Callback (function pointer)
int (*soap.fsslauth)(**struct** soap *soap)
Initialize the authentication information for clients and services, such as the certificate chain, password, read the key and/or DH file, generate an RSA key, and initialization of the RNG. Should return a gSOAP error code or SOAP_OK. Built-in gSOAP function: ssl_auth_init

The fsslverify callback function controls OpenSSL peer certificate verification, via internally invoking SSL_CTX_set_verify:

Callback (function pointer)
int (*soap.fssverify)(**int** ok, X509_STORE_CTX *store
Used to control the certificate verification behaviour when the SOAP_SSL_REQUIRE_CLIENT_AUTHENTICATION or SOAP_SSL_REQUIRE_SERVER_AUTHENTICATION flags are specified with soap_ssl_client_context and soap_ssl_server_context. It receives two arguments: ok indicates, whether the verification of the certificate in question was passed (ok=1) or not (]blkok=0). If the callback returns 0, the verification process is immediately stopped with "verification failed" state. A verification failure alert is sent to the peer and the TLS/SSL handshake is terminated. If the callback returns 1, the verification process is continued. Built-in gSOAP function: ssl_verify_callback and ssl_verify_callback_allow_expired_certificate. These functions are used when fsslverify is initially set to NULL and were not reassigned before soap_ssl_client_context or soap_ssl_server_context are called.

19.24   SSL Certificates and Key Files

The gSOAP distribution includes a cacerts.pem file with the certificates of all certificate authorities (such as Verisign). You can use this file to verify the authentication of servers that provide certificates issued by these CAs. Just set the cafile parameter to the location of this file on your file system. Therefore, when you obtain a certifice signed by a trusted CA such as Verisign, you can simply use the cacerts.pem file to develop client applications that can verify the authenticity of your server.

Althernatively, you can use the plugin/cacerts.h and plugin/cacerts.c code to embed CA certificates in your client code.

For systems based on Microsoft windows, the WinInet module can be used instead, see the README.txt located in the package under mod_gsoap/gsoap_win/wininet.

The other .pem files in the gSOAP distribution are examples of self-signed certificates for testing purposes (cacert.pem, client.pem, server.pem). The client.pem and server.pem contain the private key and certificate of the client or server, respectively. The keyfiles (client.pem and server.pem) are created by concatenating the private key PEM with the certificate PEM. The keyfile SHOULD NEVER be shared with any party. With OpenSSL, you can encrypt the keyfiles with a password to offer some protection and the password is used in the client/server code to read the keyfile. GNUTLS does not support this feature and cannot encrypt or decrypt a keyfile.

You can also create your own self-signed certificates. There is more than one way to generate the necessary files for clients and servers. See http://www.openssl.org for information on OpenSSL and http://sial.org/howto/openssl/ca/ on how to setup and manage a local CA and http://sial.org/howto/openssl/self-signed/ on how to setup self-signed test certificates.

It is possible to convert IIS-generated certificates to PEM format with the openssl library and openssl command-line tool:

```
openssl x509 -in mycert.cer -inform DER -out mycert.pem -outform PEM
```

This converts the CRT-formatted mycert.cer to PEM-formatted mycert.pem.

Here is the simplest way to setup self-signed certificates. First you need to create a private Certificate Authority (CA). The CA is used in SSL to verify the authenticity of a given certificate.

The CA acts as a trusted third party who has authenticated the user of the signed certificate as being who they say. The certificate is signed by the CA, and if the client trusts the CA, it will trust your certificate. For use within your organization, a private CA will probably serve your needs. However, if you intend use your certificates for a public service, you should probably obtain a certificate from a known CA (e.g. VeriSign). In addition to identification, your certificate is also used for encryption.

Creating certificates should be done through a CA to obtain signed certificates. But you can create your own certificates for testing purposes as follows.

- Go to the OpenSSL bin directory (/usr/local/ssl by default and /System/Library/OpenSSL on Mac OS X)

- There should be a file called openssl.cnf

- Create a new directory in your home account, e.g. $HOME/CA, and copy the openssl.cnf file to this directory

- Modify openssl.cnf by changing the 'dir' value to HOME/CA

- Copy the README.txt, root.sh, and cert.sh scripts from the gSOAP distribution package located in the samples/ssl directory to HOME/CA

- Follow the README.txt instructions

You now have a self-signed CA root certificate cacert.pem and a server.pem (or client.pem) certificate in PEM format. The cacert.pem certificate is used in the cafile parameter of the soap_ssl_client_context (or soap_ssl_server_context) at the client (or server) side to verify the authenticity of the peer. You can also provide a capath parameter to these trusted certificates. The server.pem (or client.pem) must be provided with the soap_ssl_server_context at the server side (or soap_ssl_client_context at the client side) together with the password you entered when generating the certificate using cert.sh to access the file. These certificates must be present to grant authentication requests by peers. In addition, the server.pem (and client.pem) include the host name of the machine on which the application runs (e.g. localhost), so you need to generate new certificates when migrating a server (or client).

Finally, you need to generate Diffie-Helmann (DH) parameters for the server if you wish to use DH instead of RSA. There are two options:

1. Set the dhfile parameter to the numeric DH prime length in bits required (for example "1024") to let the engine generate DH parameters at initialization. This can be time consuming.

2. Provide a file name for the dhfile parameter of soap_ssl_server_context. The file should be generated beforehand. To do so with the OpenSSL command line tool, use:

    > openssl dhparam -outform PEM -out dh.pem 512

   File dh512.pem is the output file and 512 is the number of bits used.

19.25 SSL Hardware Acceleration

You can specify a hardware engine to enable hardware support for cryptographic acceleration. This can be done once in a server or client with the following statements:

```
static const char *engine = "cswift"; /* engine name */
int main()
{
  ...
  ENGINE *e;
  if (!(e = ENGINE_by_id(engine)))
    fprintf(stderr, "Error finding engine %s\n", engine);
  else if (!ENGINE_set_default(e, ENGINE_METHOD_ALL))
    fprintf(stderr, "Error using engine %s\n", engine);
  ...
```

The following table lists the names of the hardware and software engines:

Name	Description
openssl	The default software engine for cryptographic operations
openbsd_dev_crypto	OpenBSD supports kernel level cryptography
cswift	CryptoSwift acceleration hardware
chil	nCipher CHIL acceleration hardware
atalla	Compaq Atalla acceleration hardware
nuron	Nuron acceleration hardware
ubsec	Broadcom uBSec acceleration hardware
aep	Aep acceleration hardware
sureware	SureWare acceleration hardware

19.26 SSL on Windows

Set the full path to libssl.lib and libcrypto.lib under the MSVC++ "Projects" menu, then choose "Link": "Object/Modules". Please make sure libssl32.dll and libeay32.dll can be loaded by gSOAP applications, thus they must be installed properly on the target machine.

If you're using compilation settings such as /MTd then link to the correct libeay32MTd.lib and ssleay32MTd.lib libraries.

Alternatively, you can use the WinInet interface available in the mod_gsoap directory of the gSOAP package. API instructions are included in the source.

19.27 Zlib Compression

To enable deflate and gzip compression with Zlib, install Zlib from http://www.zlib.org if not already installed on your system. Compile stdsoap2.cpp (or stdsoap2.c) and **all** your sources that include stdsoap2.h or soapH.h with compiler option -DWITH_GZIP and link your code with the Zlib library, e.g. -lz on Unix/Linux platforms.

The gzip compression is orthogonal to all transport encodings such as HTTP, SSL, DIME, and can be used with other transport layers. You can even save and load compressed XML data to/from files.

gSOAP supports two compression formats: deflate and gzip. The gzip format is used by default. The gzip format has several benefits over deflate. Firstly, gSOAP can automatically detect gzip compressed inbound messages, even without HTTP headers, by checking for the presence of a gzip header in the message content. Secondly, gzip includes a CRC32 checksum to ensure messages have been correctly received. Thirdly, gzip compressed content can be decompressed with other compression software, so you can decompress XML data saved by gSOAP in gzip format.

Gzip compression is enabled by compiling the sources with -DWITH_GZIP. To transmit gzip compressed SOAP/XML data, set the output mode flags to SOAP_ENC_ZLIB. For example:

```
soap_init(&soap);
...
soap_set_omode(&soap, SOAP_ENC_ZLIB); // enable Zlib's gzip
if (soap_call_ns__myMethod(&soap, ...))
...
soap_clr_omode(&soap, SOAP_ENC_ZLIB); // disable Zlib's gzip
...
```

This will send a compressed SOAP/XML request to a service, provided that Zlib is installed and linked with the application and the -DWITH_GZIP option was used to compile the sources. Receiving compressed SOAP/XML over HTTP either in gzip or deflate formats is automatic. The SOAP_ENC_ZLIB flag does not have to be set at the server side to accept compressed messages. Reading and receiving gzip compressed SOAP/XML without HTTP headers (e.g. with other transport protocols) is also automatic.

To control the level of compression for outbound messages, you can set the soap.z_level to a value between 1 and 9, where 1 is the best speed and 9 is the best compression (default is 6). For example

```
soap_init(&soap);
...
soap_set_omode(&soap, SOAP_ENC_ZLIB);
soap.z_level = 9; // best compression
...
```

To verify and monitor compression rates, you can use the values soap.z_ratio_in and soap.z_ratio_out. These two float values lie between 0.0 and 1.0 and express the ratio of the compressed message length over uncompressed message length.

```
soap_call_ns__myMethod(&soap, ...);
...
printf("Compression ratio: %f%% (in) %f%% (out)\n", 100*soap.z_ratio_out, 100*soap.z_ratio_in);
...
```

Note: lower ratios mean higher compression rates.

Compressed transfers require buffering the entire output message to determine HTTP message length. This means that the SOAP_IO_STORE flag is automatically set when the SOAP_ENC_ZLIB flag

is set to send compressed messages. The use of HTTP chunking significantly reduces memory usage and may speed up the transmission of compressed SOAP/XML messages. This is accomplished by setting the SOAP_IO_CHUNK flag with SOAP_ENC_ZLIB for the output mode. However, some Web servers do not accept HTTP chunked request messages (even when they return HTTP chunked messages!). Stand-alone gSOAP services always accept chunked request messages.

To restrict the compression to the deflate format only, compile the sources with -DWITH_ZLIB. This limits compression and decompression to the deflate format. Only plain and deflated messages can be exchanged, gzip is not supported with this option. Receiving gzip compressed content is automatic, even in the absence of HTTP headers. Receiving deflate compressed content is not automatic in the absence of HTTP headers and requires the flag SOAP_ENC_ZLIB to be set for the input mode to decompress deflated data.

Caution: it is important that the WITH_GZIP and WITH_ZLIB macros MUST be consistently defined to compile the sources, such as stdsoap2.cpp, soapC.cpp, soapClient.cpp, soapServer.cpp, and all application sources that include stdsoap2.h or soapH.h. If the macros are not consistently used, the application will crash due to a mismatches in the declaration and access of the gSOAP context.

19.28   Client-Side Cookie Support

Client-side cookie support is optional. To enable cookie support, compile all sources with option -DWITH_COOKIES, for example:

> c++ -DWITH_COOKIES -o myclient stdsoap2.cpp soapC.cpp soapClient.cpp

or add the following line to stdsoap.h:

#define WITH_COOKIES

Client-side cookie support is fully automatic. So just (re)compile stdsoap2.cpp with -DWITH_COOKIES to enable cookie-based session control in your client.

A database of cookies is kept and returned to the appropriate servers. Cookies are not automatically saved to a file by a client. An example cookie file manager is included as an extras in the distribution. You should explicitly remove all cookies before terminating a gSOAP context by calling soap_free_cookies(soap) or by calling soap_done(soap).

To avoid "cookie storms" caused by malicious servers that return an unreasonable amount of cookies, gSOAP clients/servers are restricted to a database size that the user can limit (32 cookies by default), for example:

```
struct soap soap;
soap_init(&soap);
soap.cookie_max = 10;
```

The cookie database is a linked list pointed to by soap.cookies where each node is declared as:

```
struct soap_cookie
{
```

```
        char *name;
        char *value;
        char *domain;
        char *path;
        long expire; /* client-side: local time to expire; server-side: seconds to expire */
        unsigned int version;
        short secure;
        short session; /* server-side */
        short env; /* server-side: 1 = got cookie from client */
        short modified; /* server-side: 1 = client cookie was modified */
        struct soap_cookie *next;
    };
```

Since the cookie database is linked to a soap struct, each thread has a local cookie database in a multi-threaded implementation.

19.29 Server-Side Cookie Support

Server-side cookie support is optional. To enable cookie support, compile all sources with option -DWITH_COOKIES, for example:

> c++ -DWITH_COOKIES -o myserver ...

gSOAP provides the following cookie API for server-side cookie session control:

Function

struct soap_cookie *soap_set_cookie(**struct** soap *soap, **const char** *name, **const char** *value, **const char** *domain, **const char** *path);
Add a cookie to the database with name name and value value. domain and path may be NULL to use the current domain and path given by soap_cookie_domain and soap_cookie_path. If successful, returns pointer to a cookie node in the linked list, or NULL otherwise.

struct soap_cookie *soap_cookie(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path);
Find a cookie in the database with name name and value value. domain and path may be NULL to use the current domain and path given by soap_cookie_domain and soap_cookie_path. If successful, returns pointer to a cookie node in the linked list, or NULL otherwise.

char *soap_cookie_value(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path);
Get value of a cookie in the database with name name. domain and path may be NULL to use the current domain and path given by soap_cookie_domain and soap_cookie_path. If successful, returns the string pointer to the value, or NULL otherwise.

long soap_cookie_expire(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path);
Get expiration value of the cookie in the database with name name (in seconds). domain and path may be NULL to use the current domain and path given by soap_cookie_domain and soap_cookie_path. Returns the expiration value, or -1 if cookie does not exist.

int soap_set_cookie_expire(**struct** soap *soap, **const char** *name, **long** expire, **const char** *domain, **const char** *path);
Set expiration value expire of the cookie in the database with name name (in seconds). domain and path may be NULL to use the current domain and path given by soap_cookie_domain and soap_cookie_path. If successful, returns SOAP_OK, or SOAP_EOF otherwise.

int soap_set_cookie_session(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path);
Set cookie in the database with name name to be a session cookie. This means that the cookie will be returned to the client. (Only cookies that are modified are returned to the client). domain and path may be NULL to use the current domain and path given by soap_cookie_domain and soap_cookie_path. If successful, returns SOAP_OK, or SOAP_EOF otherwise.

int soap_clr_cookie_session(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path);
Clear cookie in the database with name name to be a session cookie. domain and path may be NULL to use the current domain and path given by soap_cookie_domain and soap_cookie_path. If successful, returns SOAP_OK, or SOAP_EOF otherwise.

void soap_clr_cookie(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path);
Remove cookie from the database with name name. domain and path may be NULL to use the current domain and path given by soap_cookie_domain and soap_cookie_path.

int soap_getenv_cookies(**struct** soap *soap);
Initializes cookie database by reading the 'HTTP_COOKIE' environment variable. This provides a means for a CGI application to read cookies send by a client. If successful, returns SOAP_OK, or SOAP_EOF otherwise.

void soap_free_cookies(**struct** soap *soap);
Release cookie database.

The following global variables are used to define the current domain and path:

Attribute	value
const char *cookie_domain	MUST be set to the domain (host) of the service
const char *cookie_path	MAY be set to the default path to the service
int cookie_max	maximum cookie database size (default=32)

The cookie_path value is used to filter cookies intended for this service according to the path prefix rules outlined in RFC2109.

The following example server adopts cookies for session control:

```
int main()
{
    struct soap soap;
    int m, s;
    soap_init(&soap);
    soap.cookie_domain = "...";
    soap.cookie_path = "/"; // the path which is used to filter/set cookies with this destination
    if (argc < 2)
    {
        soap_getenv_cookies(&soap); // CGI app: grab cookies from 'HTTP_COOKIE' env var
        soap_serve(&soap);
    }
    else
    {
        m = soap_bind(&soap, NULL, atoi(argv[1]), 100);
        if (m < 0)
            exit(1);
        for (int i = 1; ; i++)
        {
            s = soap_accept(&soap);
            if (s < 0)
                exit(1);
            soap_serve(&soap);
            soap_end(&soap); // clean up
            soap_free_cookies(&soap); // remove all old cookies from database so no interference occurs
with the arrival of new cookies
        }
    }
    return 0;
}
int ck__demo(struct soap *soap, ...)
{
    int n;
    const char *s;
    s = soap_cookie_value(soap, "demo", NULL, NULL); // cookie returned by client?
    if (!s)
        s = "init-value"; // no: set initial cookie value
    else
        ... // modify 's' to reflect session control
    soap_set_cookie(soap, "demo", s, NULL, NULL);
    soap_set_cookie_expire(soap, "demo", 5, NULL, NULL); // cookie may expire at client-side in 5
seconds
    return SOAP_OK;
}
```

19.30   Connecting Clients Through Proxy Servers

When a client needs to connect to a Web Service through a proxy server, set the soap.proxy_host string and soap.proxy_port integer attributes of the current soap runtime context to the proxy's host name and port, respectively. For example:

```
struct soap soap;
soap_init(&soap);
soap.proxy_host = "proxyhostname";
soap.proxy_port = 8080;
if (soap_call_ns__method(&soap, "http://host:port/path", "action", ...))
   soap_print_fault(&soap, stderr);
else
   ...
```

The attributes soap.proxy_host and soap.proxy_port keep their values through a sequence of service operation calls, so they only need to be set once.

When X-Forwarded-For headers are returned by the proxy, the header can be accessed in the soap.proxy_from string.

19.31   FastCGI Support

To enable FastCGI support, install FastCGI and compile *all* sources (do not use libgsoap but compile stdsoap2.c) and your application sources with option -DWITH_FASTCGI or add

    #define WITH_FASTCGI

to stdsoap2.h and recompile the project code.

Caution: Do not link against the libgsoap libraries as these are not suitable for FastCGI. Compile stdsoap2.c (or stdsoap2.cpp) instead.

19.32   How to Create gSOAP Applications With a Small Memory Footprint

To compile gSOAP applications intended for small memory devices, you may want to remove all non-essential features that consume precious code and data space. To do this, compile the gSOAP sources with -DWITH_LEAN (i.e. #define WITH_LEAN) to remove many non-essential features. The features that will be disabled are:

- No I/O timeouts. Note that many socket operations already obey some form of timeout handling, such as a connect timeout for example.

- No UDP support

- No HTTP keep alive

- No HTTP cookies

- No HTTP authentication

- No HTTP chunked output (but input is OK)

- No HTTP compressed output (but input is OK when compiled with WITH_GZIP)

- No send/recv timeouts

- No socket flags (no soap.socket_flag, soap.connect_flag, soap.bind_flag, soap.accept_flag)

- No canonical XML output

- No logging

- Limited TCP/IP and HTTP error diagnostic messages

- No support for time_t serialization

- No support for hexBinary XML attribute serialization (remap hexBinary to strings by adding a remap entry to typemap.dat)

Use -DWITH_LEANER to make the executable even smaller by removing DIME and MIME attachment handling, LONG64 (64 bit) serialization, wchar_t* serialization, and support for XML DOM operations. Note that DIME/MIME attachments are not essential to achieve SOAP/XML interoperability. DIME attachments are a convenient way to exchange non-text-based (i.e. binary) content, but are not required for basic SOAP/XML interoperability. Attachment requirements are predictable. That is, applications won't suddenly decide to use DIME or MIME instead of XML to exchange content.

It is safe to try to compile your application with -DWITH_LEAN, provided that your application does not rely on I/O timeouts. When no linkage error occurs in the compilation process, it is safe to assume that your application will run just fine.

19.33   How to Eliminate BSD Socket Library Linkage

The stdsoap2.c and stdsoap2.cpp gSOAP runtime libraries should be linked with a BSD socket library in the project build, e.g. winsock2 for Win32. To eliminate the need to link a socket library, you can compile stdsoap2.c (for C) and stdsoap2.cpp (for C++) with the -DWITH_NOIO macro set (i.e. #define WITH_NOIO). This eliminates the dependency on the BSD socket API, IO streams, FILE type, and errno.

As a consequence, you MUST define callbacks to replace the missing socket stack. To do so, add to your code the following definitions:

```
struct soap soap;
soap_init(&soap);
/* fsend is used to transmit data in blocks */
soap.fsend = my_send;
/* frecv is used to receive data in blocks */
soap.frecv = my_recv;
```

```
/* fopen is used to connect */
soap.fopen = my_tcp_connect;
/* fclose is used to disconnect */
soap.fclose = my_tcp_disconnect;
/* fclosesocket is used only to close the master socket in a server upon soap_done() */
soap.fclosesocket = my_tcp_closesocket;
/* fshutdownsocket is used after completing a send operation to send TCP FIN */
soap.fshutdownsocket = my_tcp_shutdownsocket;
/* setting fpoll is optional, leave it NULL to omit polling the server */
soap.fpoll = my_poll;
/* faccept is used only by a server application */
soap.faccept = my_accept;
```

These functions are supposed to provide a (minimal) transport stack. See Section 19.7 for more details on the use of these callbacks. All callback function pointers should be non-NULL, except fpoll.

You cannot use soap_print_fault and soap_print_fault_location to print error diagnostics. Instead, the value of soap.error, which contains the gSOAP error code, can be used to determine the cause of a fault.

19.34 How to Combine Multiple Client and Server Implementations into one Executable

The wsdl2h tool can be used to import multiple WSDLs and schemas at once. The service definitions are combined in one header file to be parsed by soapcpp2. It is important to assign namespace prefixes to namespace URIs using the typemap.dat file. Otherwise, wsdl2h will assign namespace prefixes ns1, ns2, and so on to the service operations and schema types. Thus, any change to a WSDL or schema may result in a new prefix assignment. For more details, please see Section 8.2.

Another approach to combine multiple client and service applications into one executable is by using C++ namespaces to structurally separate the definitions or by creating C libraries for the client/server objects as explained in subsequent sections. This is automated with wsdl2h option -q.

Both approaches are demonstrated by example in the gSOAP distribution, the samples/link (C only) and samples/link++ (C++ with C++ namespaces) examples.

19.35 How to Build a Client or Server in a C++ Code Namespace

You can use a C++ code namespace of your choice in your header file to build a client or server in that code namespace. In this way, you can create multiple clients and servers that can be combined and linked together without conflicts, which is explained in more detail in the next section (which also shows an example combining two client libraries defined in two C++ code namespaces).

Use wsdl2h option -q*name* to generate definitions in the C++ *name* namespace. This option can also be used in combination with C++ proxy and server object generation, using soapcpp2 options -i (or -j) and -p.

At most one namespace can be defined for the entire gSOAP header file. The code namespace MUST completely encapsulate the entire contents of the header file:

```
namespace myNamespaceName {
... gSOAP header file contents ...
}
```

When compiling this header file with the gSOAP soapcpp2 compiler, all type definitions, the
(de)serializers for these types, and the stub/skeleton codes will be placed in this namespace. The
XML namespace mapping table (saved in a .nsmap file) will not be placed in the code namespace
to allow it to be linked as a global object. You can use option -n to create local XML namespace
tables, see Section 9.1 (but remember that you explicitly need to initialize the soap.namespaces to
point to a table at run time). The generated files are prefixed with the code namespace name
instead of the usual soap file name prefix to enable multiple client/server codes to be build in the
same project directory (a code namespace automatically sets the -p compiler option, see Section 9.1
for options).

Because the SOAP Header and Fault serialization codes will also be placed in the namespace, they
cannot be called from the stdsoap2.cpp run time library code and are therefore rendered unusable.
Therefore, these serializers are not compiled at all (enforced with #define WITH_NOGLOBAL). To add
SOAP Header and Fault serializers, you MUST compile them separately as follows. First, create a
new header file env.h with the SOAP Header and Fault definitions. You can leave this header file
empty if you want to use the default SOAP Header and Fault. Then compile this header file with:

> soapcpp2 -penv env.h

The generated envC.cpp file holds the SOAP Header and Fault serializers and you can link this file
with your client or server application.

19.36   How to Create Client/Server Libraries

The gSOAP soapcpp2 compiler produces soapClientLib.cpp and soapServerLib.cpp codes that are specif-
ically intended for building static or dynamic client/server libraries. These codes export the stubs
and skeletons, but keep all marshaling code (i.e. parameter serializers and deserializers) local (i.e. as
static functions) to avoid link symbol conflicts when combining multiple clients and/or servers into
one executable. Note that it is far simpler to use the wsdl2h tool on multiple WSDL files to generate
a header file that combines all service definitions. However, the approach presented in this section
is useful when creating (dynamic) libraries for client and server objects, such as DLLs as described
in Section 19.37.

Do not link soapClientLib.cpp or soapServerLib.cpp together with soapC.cpp, soapClient.cpp, and soapServer.cpp.
The library versions already include all of the necessary definitions.

To build multiple libraries in the same project directory, you can define a C++ code namespace in
your header file (see Section 19.35) or you can use soapcpp2 with option -p to rename the generated
soapClientLib.cpp and soapServerLib.cpp (and associated) files. The -p option specifies the file name
prefix to replace the soap prefix. The libraries don't have to be C++ codes. You can use option -c to
generate C code. A clean separation of libraries can also be achieved with C++ code namespaces,
see Section 19.35.

The library codes do not define SOAP Header and Fault serializers. You MUST add SOAP Header
and Fault serializers to your application, which are compiled separately as follows. First, create a

new header file env.h with the SOAP Header and Fault definitions. You can leave this header file empty if you want to use the default SOAP Header and Fault. Then compile this header file with:

```
> soapcpp2 -penv env.h
```

The generated envC.cpp file holds the SOAP Header and Fault serializers and you can create a (dynamic) library for it to link this code with your client or server application.

You MUST compile the stdsoap2.cpp library using -DWITH_NONAMESPACES:

```
> c++ -DWITH_NONAMESPACES -c stdsoap2.cpp
```

This omits the reference to the global namespaces table, which is nowhere to be defined since we will use XML namespaces for each client/service separately. Therefore, you MUST explicitly set the namespaces value of the gSOAP context in your code every time after initialization of the soap struct with the soap_set_namespaces(**struct** soap*, **const struct** Namespace*) function.

For example, suppose we have two clients defined in header files client1.h and client2.h. We first generate the envH.h file for the SOAP Header and Fault definitions:

```
> soapcpp2 -c -penv env.h
```

Then we generate the code for client1 and client2:

```
> soapcpp2 -c -n -pmyClient1 client1.h
> soapcpp2 -c -n -pmyClient2 client2.h
```

This generates myClient1ClientLib.c and myClient2ClientLib.c (among many other files). These two files should be compiled and linked with your application. The source code of your application should include the generated envH.h, myClient1H.h, myClient2.h files and myClient1.nsmap, myClient2.nsmap files:

```
#include "envH.h" // include this file first!
#include "myClient1H.h" // include client 1 stubs
#include "myClient2H.h" // include client 2 stubs
...
#include "myClient1H.nsmap" // include client 1 nsmap
#include "myClient2H.nsmap" // include client 2 nsmap
...
soap_init(&soap);
soap_set_namespaces(&soap, myClient1_namespaces);
... make Client 1 invocations ...
...
soap_set_namespaces(&soap, myClient2_namespaces);
... make Client 2 invocations ...
```

It is important to use soapcpp2 option -n, see Section 9.1, to rename the namespace tables so we can include them all without running into redefinitions.

Note: Link conflicts may still occur in the unlikely situation that identical service operation names are defined in two or more client stubs or server skeletons when these methods share the same XML namespace prefix. You may have to use C++ code namespaces to avoid these link conflicts or rename the namespace prefixes used by the service operation defined in the header files.

269

19.36.1 C++ Clients Example

As an example we will build a Delayed Stock Quote client library and a Currency Exchange Rate client library.

First, we create an empty header file env.h (which may contain optional SOAP Header and Fault definitions), and compile it as follows:

```
> soapcpp2 -penv env.h
> c++ -c envC.cpp
```

We also compile stdsoap2.cpp without namespaces:

```
> c++ -c -DWITH_NONAMESPACES stdsoap2.cpp
```

Note: when you forget to use -DWITH_NONAMESPACES you will get an unresolved link error for the global namespaces table. You can define a dummy table to avoid having to recompile stdsoap2.cpp.

Second, we create the Delayed Stock Quote header file specification, which may be obtained using the WSDL importer. If you want to use C++ namespaces then you need to manually add the **namespace** declaration to the generated header file:

```
namespace quote {
//gsoap ns service name: Service
//gsoap ns service style: rpc
//gsoap ns service encoding: encoded
//gsoap ns service location: http://services.xmethods.net/soap
//gsoap ns schema namespace: urn:xmethods-delayed-quotes
//gsoap ns service method-action: getQuote ""
int ns__getQuote(char *symbol, float &Result);
}
```

We then compile it as a library and we use option -n to rename the namespace table to avoid link conflicts later:

```
> soapcpp2 -n quote.h
> c++ -c quoteClientLib.cpp
```

If you don't want to use a C++ code namespace, you should compile quote.h "as is" with soapcpp2 option -pquote:

```
> soapcpp2 -n -pquote quote.h
> c++ -c quoteClientLib.cpp
```

Third, we create the Currency Exchange Rate header file specification:

```
namespace rate {
//gsoap ns service name: Service
//gsoap ns service style: rpc
```

```
//gsoap ns service encoding: encoded
//gsoap ns service location: http://services.xmethods.net/soap
//gsoap ns schema namespace: urn:xmethods-CurrencyExchange
//gsoap ns service method-action: getRate ""
int ns__getRate(char *country1, char *country2, float &Result);
}
```

Similar to the Quote example above, we compile it as a library and we use option -n to rename the namespace table to avoid link conflicts:

```
> soapcpp2 -n rate.h
```

Fourth, we consider linking the libraries to the main program. The main program can import the quoteServiceProxy.h and rateServiceProxy.h files to obtain client proxies to invoke the services. The proxy implementations are defined in quoteClient.cpp. The -n option also affects the generation of the C++ proxy codes to ensure that the gSOAP context is properly initialized with the appropriate namespace table (so you don't have to initialize explicitly – this feature is only available with C++ proxy and server object classes).

```
#include "quoteServiceProxy.h" // get quote Service proxy
#include "rateServiceProxy.h" // get rate Service proxy
#include "quote.nsmap" // get quote namespace bindings
#include "rate.nsmap" // get rate namespace bindings
int main(int argc, char *argv[])
{
  if (argc <= 1)
  {
    std::cerr << "Usage: main ticker [currency]" << std::endl
    exit(0);
  }
  quote::Service quote;
  float q;
  if (quote.getQuote(argv[1], q)) // get quote
    soap_print_fault(quote.soap, stderr);
  else
  {
    if (argc > 2)
    {
      rate::Service rate;
      float r;
      if (rate.getRate("us", argv[2], r)) // get rate in US dollars
        soap_print_fault(rate.soap, stderr);
      else
        q *= r; // convert the quote
    }
    std::cout << argv[1] << ": " << q << std::endl;
  }
  return 0;
}
```

Compile and link this application with stdsoap2.o, envC.o, quoteServerProxy.o, and rateServerProxy.o.

To compile and link a server object is very similar. For example, assume that we need to implement a calculator service and we want to create a library for it.

```
namespace calc {
//gsoap ns service name: Service
//gsoap ns service style: rpc
//gsoap ns service encoding: encoded
//gsoap ns service location: http://www.cs.fsu.edu/~engelen/calc.cgi
//gsoap ns schema namespace: urn:calc
int ns__add(double a, double b, double &result);
int ns__sub(double a, double b, double &result);
int ns__mul(double a, double b, double &result);
int ns__div(double a, double b, double &result);
}
```

We compile this with:

```
> soapcpp2 -n calc.h
```

The effect of the -n option is that it creates local namespace tables, and a modified calcServiceObject.h server class definitions that properly initialize the gSOAP run time with the table.

```
#include "calcServiceObject.h" // get Service object
#include "calc.nsmap" // get calc namespace bindings
...
calc::Service calc;
calc.serve(); // calls request dispatcher to invoke one of the functions below
...
int calc::Service::add(double a, double b, double &result);
{ result = a + b; returnSOAP_OK; }
int calc::Service::sub(double a, double b, double &result);
{ result = a - b; returnSOAP_OK; }
int calc::Service::mul(double a, double b, double &result);
{ result = a * b; returnSOAP_OK; }
int calc::Service::div(double a, double b, double &result);
{ result = a / b; returnSOAP_OK; }
```

In fact, the calc::Service class is derived from the **struct** soap. So the context is available as this, which can be passed to all gSOAP functions that require a soap struct context.

The example above serves requests over stdin/out. Use the bind and accept calls to create a stand-alone server to service inbound requests over sockets, see 7.2.3.

19.36.2   C Clients Example

This is the same example as above, but the clients are build with pure C.

We create a env.h that contains the joint SOAP Header and SOAP Fault definitions. You may have to copy-paste these from the other header files. Then, compile it as follows:

```
> soapcpp2 -c -penv env.h
> cc -c envC.c
```

We also compile stdsoap2.c without namespaces:

```
> cc -c -DWITH_NONAMESPACES stdsoap2.c
```

Second, we create the Delayed Stock Quote header file specification, which may be obtained using the WSDL importer.

```
//gsoap ns service name: Service
//gsoap ns service style: rpc
//gsoap ns service encoding: encoded
//gsoap ns service location: http://services.xmethods.net/soap
//gsoap ns schema namespace: urn:xmethods-delayed-quotes
//gsoap ns service method-action: getQuote ""
int ns__getQuote(char *symbol, float *Result);
```

We compile it as a library and we use options -n and -p to rename the namespace table to avoid link conflicts:

```
> soapcpp2 -c -n -pquote quote.h
> cc -c quoteClientLib.c
```

Third, we create the Currency Exchange Rate header file specification:

```
//gsoap ns service name: Service
//gsoap ns service style: rpc
//gsoap ns service encoding: encoded
//gsoap ns service location: http://services.xmethods.net/soap
//gsoap ns schema namespace: urn:xmethods-CurrencyExchange
//gsoap ns service method-action: getRate ""
int ns__getRate(char *country1, char *country2, float *Result);
```

We compile it as a library and we use options -n and -p to rename the namespace table to avoid link conflicts:

```
> soapcpp2 -c -n -prate rate.h
> cc -c rateClientLib.c
```

The main program is:

```
#include "quoteStub.h" // get quote Service stub
#include "rateStub.h" // get rate Service stub
#include "quote.nsmap" // get quote namespace bindings
```

273

```
#include "rate.nsmap" // get rate namespace bindings
int main(int argc, char *argv[])
{
  if (argc <= 1)
  {
    fprintf(stderr, "Usage: main ticker [currency]\n");
    exit(0);
  }
  struct soap soap;
  float q;
  soap_init(&soap);
  soap_set_namespaces(&soap, quote_namespaces);
  if (soap_call_ns__getQuote(&soap, "http://services.xmethods.net/soap", "", argv[1], &q)) // get
quote
    soap_print_fault(&soap, stderr);
  else
  {
    if (argc > 2)
    {
      soap_set_namespaces(&soap, rate_namespaces);
      float r;
      if (soap_call_ns__getRate(&soap, "http://services.xmethods.net/soap", "", "us", argv[2],
&r)) // get rate in US dollars
        soap_print_fault(&soap, stderr);
      else
        q *= r; // convert the quote
    }
    printf("%s: %f \n", argv[1], q);
  }
  return 0;
}
```

Compile and link this application with stdsoap2.o, envC.o, quoteClientLib.o, and rateClientLib.o.

To compile and link a server library is very similar. Assuming that the server is named "calc" (as specified with options -n and -p), the application needs to include the calcStub.h file, link the calcServerLib.o file, and call calc_serve(&soap) function at run time.

19.36.3 C Services Chaining Example

We build a C application for multiple services served on one port.

We create a env.h that contains the joint SOAP Header and SOAP Fault definitions. You may have to copy-paste these from the other header files. Then, compile it as follows:

```
> soapcpp2 -c -penv env.h
> cc -c envC.c
```

We also compile stdsoap2.c without namespaces:

```
> cc -c -DWITH_NONAMESPACES stdsoap2.c
```

Say we have a service definition in quote.h. We compile it as a library and we use options -n and -p to rename the namespace table to avoid link conflicts:

```
> soapcpp2 -c -n -pquote quote.h
> cc -c quoteClientLib.c
```

We do the same for a service definition in rate.h:

```
> soapcpp2 -c -n -prate rate.h
> cc -c rateClientLib.c
```

To serve both the quote and rate services on the same port, we chain the service dispatchers as follows:

```
struct soap *soap = soap_new();
soap_bind(soap, NULL, 8080, 100);
soap_accept(soap);
if (soap_begin_serve(soap))
  soap_send_fault(&abc); // send fault to client
else if (quote_serve_request(soap) == SOAP_NO_METHOD)
{
  if (rate_serve_request(soap))      soap_send_fault(soap); // send fault to client
}
else if (soap.error)
  soap_send_fault(soap); // send fault to client
soap_destroy(soap);
soap_end(soap);
soap_free(soap);
```

This chaining can be arbitrarily deep. When the previous request fails with a SOAP_NO_METHOD then next request dispatcher can be tried.

The server should also define the service operations:

```
int ns__getQuote(struct soap *soap, char *symbol, float *Result);
{ *Result = ... ;
  return SOAP_OK;
}
int ns__getRate(struct soap *soap, char *country1, char *country2, float *Result);
{ *Result = ... ;
  return SOAP_OK;
}
```

19.37   How to Create DLLs

19.37.1   Create the Base stdsoap2.dll

First, create a new header file env.h with the SOAP Header and Fault definitions. You can leave this header file empty if you want to use the default SOAP Header and Fault. Then compile this header file with:

```
> soapcpp2 -penv env.h
```

The generated envC.cpp file holds the SOAP Header and Fault serializers, which need to be part of the base library functions.

The next step is to create stdsoap2.dll which consists of the file stdsoap2.cpp and envC.cpp. This DLL contains all common functions needed for all other clients and servers based on gSOAP. Compile envC.cpp and stdsoap2.cpp into stdsoap2.dll using the C++ compiler option -DWITH_NONAMESPACES and the MSVC Pre-Processor definitions SOAP_FMAC1=_ _declspec(dllexport) and SOAP_FMAC3=_ _declspec(dllexport) (or you can compile with -DWITH_SOAPDEFS_H and put the macro definitions in soapdefs.h). This exports all functions which are preceded by the macro SOAP_FMAC1 in the soapcpp2.cpp source file and macro SOAP_FMAC3 in the envC.cpp source file.

19.37.2   Creating Client and Server DLLs

Compile the soapClientLib.cpp and soapServerLib.cpp sources as DLLs by using the MSVC Pre-Processor definitions SOAP_FMAC5=_ _declspec(dllexport) and SOAP_CMAC=_ _declspec(dllexport), and by using the C++ compiler option -DWITH_NONAMESPACES. This DLL links to stdsoap2.dll.

To create multiple DLLs in the same project directory, you SHOULD use option -p to rename the generated soapClientLib.cpp and soapServerLib.cpp (and associated) files. The -p option specifies the file name prefix to replace the soap prefix. A clean separation of libraries can also be achieved with C++ namespaces, see Section 19.35.

Unless you use the client proxy and server object classes (soapXProxy.h and soapXObject.h where X is the name of the service), all client and server applications MUST explicitly set the namespaces value of the gSOAP context:

```
soap_init(&soap);
soap_set_namespaces(&soap, namespaces);
```

where the namespaces[] table should be defined in the client/server source. These tables are generated in the .nsmap files. You can rename the tables using option -n, see Section 9.1.

19.38   gSOAP Plug-ins

The gSOAP plug-in feature enables a convenient extension mechanism of gSOAP capabilities. When the plug-in registers with gSOAP, it has full access to the run-time settings and the gSOAP function callbacks. Upon registry, the plug-in's local data is associated with the gSOAP run-time. By overriding gSOAP's function callbacks with the plug-in's function callbacks, the plug-in can extend gSOAP's capabilities. The local plug-in data can be accessed through a lookup function, usually invoked within a callback function to access the plug-in data. The registry and lookup functions are:

int soap_register_plugin_arg(**struct** soap *soap, **int** (*fcreate)(**struct** soap *soap, **struct** soap_plugin *p, **void** *arg), **void** *arg)
void* soap_lookup_plugin(**struct** soap*, **const char***);

Other functions that deal with plug-ins are:

```
int soap_copy(struct soap *soap);
void soap_done(struct soap *soap);
```

The soap_copy function returns a new dynamically allocated gSOAP context that is a copy of another, such that no data is shared between the copy and the original context. The soap_copy function invokes the plug-in copy callbacks to copy the plug-ins' local data. The soap_copy function returns a gSOAP error code or SOAP_OK. The soap_done function de-registers all plugin-ins, so this function should be called to cleanly terminate a gSOAP run-time context.

An example will be used to illustrate these functions. This example overrides the send and receive callbacks to copy all messages that are sent and received to the terminal (stderr).

First, we write a header file plugin.h to define the local plug-in data structure(s) and we define a global name to identify the plug-in:

```
#include "stdsoap2.h"
#define PLUGIN_ID "PLUGIN-1.0" // some name to identify plugin
struct plugin_data // local plugin data
{
   int (*fsend)(struct soap*, const char*, size_t); // to save and use send callback
   size_t (*frecv)(struct soap*, char*, size_t); // to save and use recv callback
};
int plugin(struct soap *soap, struct soap_plugin *plugin, void *arg);
```

Then, we write the plugin registry function and the callbacks:

```
#include "plugin.h"
static const char plugin_id[] = PLUGIN_ID; // the plugin id
static int plugin_init(struct soap *soap, struct plugin_data *data);
static int plugin_copy(struct soap *soap, struct soap_plugin *dst, struct soap_plugin *src);
static void plugin_delete(struct soap *soap, struct soap_plugin *p);
static int plugin_send(struct soap *soap, const char *buf, size_t len);
static size_t plugin_recv(struct soap *soap, char *buf, size_t len);
// the registry function:
int plugin(struct soap *soap, struct soap_plugin *p, void *arg)
{
   p->id = plugin_id;
   p->data = (void*)malloc(sizeof(struct plugin_data));
   p->fcopy = plugin_copy; /* optional: when set the plugin must copy its local data */
   p->fdelete = plugin_delete;
   if (p->data)
     if (plugin_init(soap, (struct plugin_data*)p->data))
     {
       free(p->data); // error: could not init
       return SOAP_EOM; // return error
     }
   return SOAP_OK;
}
static int plugin_init(struct soap *soap, struct plugin_data *data)
```

```
{
  data->fsend = soap->fsend; // save old recv callback
  data->frecv = soap->frecv; // save old send callback
  soap->fsend = plugin_send; // replace send callback with new
  soap->frecv = plugin_recv; // replace recv callback with new
  return SOAP_OK;
}
// copy plugin data, called by soap_copy() // This is important: we need a deep copy to avoid data
sharing by two run-time contexts
static int plugin_copy(struct soap *soap, struct soap_plugin *dst, struct soap_plugin *src)
{
  if (!(dst->data = (struct plugin_data*)malloc(sizeof(struct plugin_data))))
    return SOAP_EOM;
  *dst->data = *src->data;
  return SOAP_OK;
}
// plugin deletion, called by soap_done()
static void plugin_delete(struct soap *soap, struct soap_plugin *p)
{ free(p->data); // free allocated plugin data
}
// the new send callback
static int plugin_send(struct soap *soap, const char *buf, size_t len)
{
  struct plugin_data *data = (struct plugin_data*)soap_lookup_plugin(soap, plugin_id); // fetch
plugin's local data
  fwrite(buf, len, 1, stderr); // write message to stderr
  return data->fsend(soap, buf, len); // pass data on to old send callback
}
// the new receive callback
static size_t plugin_recv(struct soap *soap, char *buf, size_t len)
{
  struct plugin_data *data = (struct plugin_data*)soap_lookup_plugin(soap, plugin_id); // fetch
plugin's local data
  size_t res = data->frecv(soap, buf, len); // get data from old recv callback
  fwrite(buf, res, 1, stderr);
  return res;
}
```

The fdelete callback of **struct** soap_plugin MUST be set to register the plugin. It is the responsibility of the plug-in to handle registry (init), copy, and deletion of the plug-in data and callbacks.

A plugin is copied with the soap_copy() call. This function copies a soap struct and the chain of plugins. It is up to the plugin implementation to share the plugin data or not:

1. if the fcopy() callback is set by the plugin initialization, this callback will be called to allow the plugin to copy its local data upon a soap_copy() call. When soap_done() is called on the soap struct copy, the fdelete() callback is called for deallocation and cleanup of the local data.

2. if the fcopy() callback is not set, then the plugin data will be shared (i.e. the data pointer points to the same address). The fdelete() callback will not be called upon a soap_done() on a copy of the soap struct. The fdelete() callback will be called for the original soap struct with

which the plugin was registered.

The example plug-in should be used as follows:

```
struct soap soap;
soap_init(&soap);
soap_register_plugin(&soap, plugin);
...
soap_done(&soap);
```

Note: soap_register_plugin(...) is an alias for soap_register_plugin_arg(..., NULL). That is, it passes NULL as an argument to plug-in's registry callback.

A number of example plug-ins are included in the gSOAP package's plugin directory. Some of these plug-ins are discussed.

19.38.1   The Message Logging and Statistics Plug-in

The message logging and access statistics plug-in can be used to selectively log inbound and outbound messages to a file or stream. It also keeps access statistics to log the total number of bytes sent and received.

To use the plug-in, compile and link your application with logging.c located in the plugin directory of the package. To enable the plug-in in your code, register the plug-in and set the streams as follows:

```
#include "logging.h"
size_t bytes_in;
size_t bytes_out;
...
if (soap_register_plugin(&soap, logging))
    soap_print_fault(&soap, stderr); // failed to register
...
soap_set_logging_inbound(&soap, stdout);
soap_set_logging_outbound(&soap, stdout);
... process messages ...
soap_set_logging_inbound(&soap, NULL); // disable logging
soap_set_logging_outbound(&soap, NULL); // disable logging
soap_get_logging_stats(&soap, &bytes_out, &bytes_in);
```

If you use soap_copy to copy the soap struct with the plug-in, the plug-in's data will be shared by the copy. Therefore, the statistics are not 100% guaranteed to be accurate for multi-threaded services since race conditions on the counters may occur. Mutex is not used to update the counters to avoid introducing expensive synchronization points. If 100% server-side accuracy is required, add mutex at the points indicated in the logging.c code.

19.38.2   RESTful Interface: The HTTP GET Plug-in

Client-side and server-side use of RESTful HTTP GET operations are supported with the HTTP GET plug-in plugin/httpget.c. The HTTP GET plug-in allows your server to handle RESTful HTTP

GET requests and at the same time still serve SOAP-based POST requests. The plug-in provides support to client applications to issue HTTP GET operations to a server.

Note that HTTP GET requests can also be handled at the server side with the fget callback, see Section 19.7. However, the HTTP GET plug-in also keeps statistics on the number of successful POST and GET exchanges and failed operations (HTTP faults, SOAP Faults, etc.). It also keeps hit histograms accumulated for up to a year of runtime.

To use the plug-in, compile and link your application with httpget.c located in the plugin directory of the package. To enable the plug-in in your code, register the plug-in with your HTTP GET handler function as follows:

```
#include "httpget.h"
...
if (soap_register_plugin_arg(&soap, httpget, (void*)my_http_get_handler))
    soap_print_fault(&soap, stderr); // failed to register
...
struct http_get_data *httpgetdata;
httpgetdata = (struct http_get_data*)soap_lookup_plugin(&soap, http_get_id);
if (!httpgetdata)
    ... // if the plug-in registered OK, there is certainly data but can't hurt to check
... process messages ...
size_t get_ok = httpgetdata->stat_get;
size_t post_ok = httpgetdata->stat_post;
size_t errors = httpgetdata->stat_fail;
...
time_t now = time(NULL);
struct tm *T;
T = localtime(&now);
size_t hitsthisminute = httpgetdata->min[T->tm_min];
size_t hitsthishour = httpgetdata->hour[T->tm_hour];
size_t hitstoday = httpgetdata->day[T->tm_yday];
```

An HTTP GET handler can simply produce some HTML content, or any other type of content by sending data:

```
int my_http_get_handler(struct *soap)
{
    soap->http_content = "text/html";
    soap_response(soap, SOAP_FILE);
    soap_send(soap, "¡html¿Hello¡/html¿");
    soap_end_send(soap);
    return SOAP_OK; // return SOAP_OK or HTTP error code, e.g. 404
}
```

If you use soap_copy to copy the soap struct with the plug-in, the plug-in's data will be shared by the copy. Therefore, the statistics are not 100% guaranteed to be accurate for multi-threaded services since race conditions on the counters may occur. Mutex is not used to update the counters to avoid introducing expensive synchronization points. If 100% server-side accuracy is required, add mutex at the points indicated in the httpget.c code.

The client-side use of HTTP GET is provided by the soap_get_connect operation. To receive a SOAP/XML (response) message ns:methodResponse, use:

```
#include "httpget.h"
...
soap_register_plugin(&soap, http_get);
...
if (soap_get_connect(&soap, endpoint, action))
   ... connect error ...
else if (soap_recv_ns_ _methodResponse(&soap, ... params ...))
   ... error ...
else
   ... ok ...
soap_destroy(&soap);
soap_end(&soap);
soap_done(&soap);
```

To receive any HTTP Body data into a buffer, use:

```
#include "httpget.h"
...
char *response = NULL;
soap_register_plugin(&soap, http_get);
...
if (soap_get_connect(&soap, endpoint, NULL))
   ... connect error ...
else if (soap_begin_recv(&soap))
   ... error ...
else
   response = soap_get_http_body(&soap);
soap_end_recv(&soap);
... use the 'response' string (NULL indicates no body or error)
soap_destroy(&soap);
soap_end(&soap);
soap_done(&soap);
```

19.38.3 RESTful Interface: The HTTP POST Plug-in

Client-side and server-side use of RESTful HTTP POST, PUT, and DELETE operations are supported with the HTTP POST plug-in plugin/httppost.c.

The HTTP POST plug-in allows your server to handle RESTful HTTP POST requests and at the same time still serve SOAP-based POST requests. The plug-in also provides support for client applications to issue HTTP POST operations to a server.

To simplify the server-side handling of POST requests, handlers can be associated with media types:

```
struct http_post_handlers my_handlers[] =
{ { "image/jpg", jpeg_handler },
  { "image/ *", image_handler },
```

```
    { "text/html", html_handler },
    { "text/ *", text_handler },
    { "text/ *;*", text_handler },
    { "POST", generic_POST_handler },
    { "PUT", generic_PUT_handler },
    { "DELETE", generic_DELETE_handler },
    { NULL }
};
```

Note that '*' can be used as a wildcard and some media types may have optional parameters (after ';'). The handlers are functions that will be invoked when a POSTed request message matching media type is send to the server.

An example image handler that checks the specific image type:

```
int image_handler(struct soap *soap)
{ const char *buf;
  size_t len;
  // if necessary, check type in soap->http_content
  if (soap->http_content && !soap_tag_cmp(soap->http_content, "image/gif")
    return 404; // HTTP error 404
  if (soap_http_body(soap, &buf, &len) != SOAP_OK)
    return soap->error;
  // ... now process image in buf
  // reply with empty HTTP OK response:
  soap_response(soap, SOAP_OK);
  soap_end_send(soap);
  return SOAP_OK;
}
```

The HTTP POST plug-in provides a soap_http_body operation as illustrated above to copy the HTTP Body content into a buffer.

The above example returns HTTP OK. If content is supposed to be returned, then use:

```
soap->http_content = "image/jpeg"; // a jpeg image to return back
soap_response(soap, SOAP_FILE); // SOAP_FILE sets custom http content
soap_send_raw(soap, buf, len); // send image
soap_end_send(soap);
```

For client applications to use HTTP POST, use the soap_post_connect operation:

```
char *buf; // holds the HTTP request/response body data
size_t len; // length of data
...
if (soap_post_connect(soap, "URL", "SOAP action or NULL", "media type")
  || soap_send_raw(soap, buf, len);
  || soap_end_send(soap))
  ... error ...
if (soap_begin_recv(&soap)
  || soap_http_body(&soap, &buf, &len)
```

```
    || soap_end_recv(&soap))
   ... error ...
// ... use buf[0..len-1]
soap_end(soap);
```

Similarly, `soap_put_connect` and `soap_delete_connect` commands are provided for PUT and DELETE handling.

19.38.4   The HTTP MD5 Checksum Plug-in

The HTTP MD5 plug-in works in the background to automatically verify the content of messages using MD5 checksums. With the plug-in, messages can be transferred over (trusted but) unreliable connections. The plug-in can be used on the client side and server side.

To use the plug-in, compile and link your application with `httpmd5.c` and `md5evp.c` located in the `plugin` directory of the package. The `md5evp.c` implementation uses the EVP interface to compute MD5 checksums with OpenSSL (compiled with -DWITH_OPENSSL).

To enable the plug-in in your code, register the plug-in as follows:

```
#include "httpmd5.h"
...
if (soap_register_plugin(&soap, http_md5))
   soap_print_fault(&soap, stderr); // failed to register
```

Once registered, MD5 checksums are produced for all outbound messages. Inbound messages with MD5 checksums in the HTTP header are automatically verified.

The plug-in requires you to set the SOAP_IO_STORE flag when sending SOAP with attachments:

```
#include "httpmd5.h"
...
struct soap soap;
soap_init1(&soap, SOAP_IO_STORE);
if (soap_register_plugin(&soap, http_md5)
   soap_print_fault(&soap, stderr); // failed to register
... now safe to send SOAP with attachments ...
```

Unfortunately, this eliminates streaming.

19.38.5   The HTTP Digest Authentication Plug-in

The HTTP digest authentication plug-in enables a more secure authentication scheme compared to basic authentication. HTTP basic authentication sends unencrypted userids and passwords over the net, while digest authentication does not exchange passwords but exchanges checksums of passwords (and other data such as nonces to avoid replay attacks). For more details, please see RFC 2617.

The HTTP digest authentication can be used next to the built-in basic authentication, or basic authentication can be rejected to tighten security. The server must have a database with userid's

and passwords (in plain text form). The client, when challenged by the server, checks the authentication realm provided by the server and sets the userid and passwords for digest authentication. The client application can temporarily store the userid and password for a sequence of message exchanges with the server, which is faster than repeated authorization challenges and authentication responses.

At the client side, the plug-in is registered and service invocations are checked for authorization challenges (HTTP error code 401). When the server challenges the client, the client should set the userid and password and retry the invocation. The client can determine the userid and password based on the authentication realm part of the server's challenge. The authentication information can be temporarily saved for multiple invocations.

Client-side example:

```
#include "httpda.h"
...
if soap_register_plugin(&soap, http_da))
   soap_print_fault(&soap, stderr); // failed to register
...
if (soap_call_ns__method(&soap, ...) != SOAP_OK)
{
   if (soap.error == 401) // challenge: HTTP authentication required
   {
      if (!strcmp(soap.authrealm, authrealm)) // determine authentication realm      {
         struct http_da_info info; // to store userid and passwd
         http_da_save(&soap, &info, authrealm, userid, passwd); // set userid and passwd for this
realm
         if (soap_call_ns__method(&soap, ...) == SOAP_OK) // retry
         { ...
           soap_end(&soap); // userid and passwd were deallocated
           http_da_restore(&soap, &info); // restore userid and passwd
           if (!soap_call_ns__method(&soap, ...) == SOAP_OK) // another call
              ...
           http_da_release(&soap, &info); // remove userid and passwd
```

This code supports both basic and digest authentication.

The server can challenge a client using HTTP code 401. With the plug-in, HTTP digest authentication challenges are send. Without the plug-in, basic authentication challenges are send.

Each server method can implement authentication as desired and may enforce digest authentication or may also accept basic authentication responses. To verify digest authentication responses, the server should compute and compare the checksums using the plug-in's http_da_verify_post function for HTTP POST requests (and http_da_verify_get for HTTP GET requests with the HTTP GET plugin) as follows:

```
#include "httpda.h"
...
if (soap_register_plugin(&soap, http_da))
   soap_print_fault(&soap, stderr); // failed to register
...
soap_serve(&soap);
```

```
    ...
    int ns__method(struct soap *soap, ...)
    {
      if (soap->userid && soap->passwd) // client used basic authentication
      { // may decide not to handle, but if ok then go ahead and compare info:
        if (!strcmp(soap->userid, userid) && !strcmp(soap->passwd, passwd))
        { ... handle request ...
          return SOAP_OK;
        }
      }
      else if (soap->authrealm && soap->userid) // Digest authentication
      {
        passwd = ... // database lookup on userid and authrealm to find passwd
        if (!strcmp(soap->authrealm, authrealm) && !strcmp(soap->userid, userid))
        {
          if (!http_da_verify_post(soap, passwd))
          { ... handle request ...
            return SOAP_OK;
          }
        }
      }
      soap->authrealm = authrealm; // set realm for challenge
      return 401; // Not authorized, challenge digest authentication
    }
```

For more details, including how to configure HTTP Digest authentication for proxies, please see the doc/httpda/html/index.html documentation in the gSOAP package.

19.38.6   The WS-Addressing Plug-in

The WSA WS-Addressing plug-in and the source code are extensively documented in the doc/wsa directory of the gSOAP package. Please refer to the documentation included in the package.

The plug-in code is located in the plugin directory:

    wsaapi.h and wsaapi.c   WS-Addressing plugin (C and C++)

To enable WS-Addressing 2005 (and support for 8/2004), the service definitions header file for soapcpp2 should include the following imports:

    #import "import/wsa5.h"

This imports the SOAP header elements required by WS-Addressing.

For more details, please see the doc/wsa/html/index.html documentation in the gSOAP package.

19.38.7   The WS-ReliableMessaging Plug-in

The WSRM WS-ReliableMessaging plug-in and the source code are extensively documented in the doc/wsrm directory of the gSOAP package. Please refer to the documentation included in the package.

The plug-in code is located in the plugin directory:

wsrmapi.h and wsrmapi.c   WS-ReliableMessaging plugin (C and C++)

Also needed are:

threads.h and threads.c   Multithreading and locking support

To enable WS-ReliableMessaging, the service definitions header file for soapcpp2 should include the following imports:

#import "import/wsrm.h"
#import "import/wsa5.h"

This imports the SOAP header elements required by WS-ReliableMessaging.

For more details, please see the doc/wsrm/html/index.html documentation in the gSOAP package.

19.38.8   The WS-Security Plug-in

The WSSE WS-Security plug-in and the source code are extensively documented in the doc/wsse directory of the gSOAP package. Please refer to the documentation included in the package for details.

The plug-in code is located in the plugin directory:

wsseapi.h and wsseapi.c   WS-Security plugin (C and C++)

Also needed are:

smdevp.h and smdevp.c   Streaming XML signature and message digest engine
mecevp.h and mecevp.c   Streaming XML encryption engine
threads.h and threads.c   Multithreading and locking support

To enable WS-Secrutiy, the service definitions header file for soapcpp2 should include the following imports:

#import "import/wsse.h"

This imports the SOAP header elements required by WS-Security.

For more details, please see the doc/wsse/html/index.html documentation in the gSOAP package.

19.38.9   WS-Discovery

The WS-Discovery implementation is documented in the doc/wsdd directory of the gSOAP package. Please refer to the documentation included in the package for details.

Basically, to add WS-Discovery support the following event handlers must be defined and linked:

286

void wsdd_event_Hello(**struct** soap *soap,
unsigned int InstanceId,
const char *SequenceId,
unsigned int MessageNumber,
const char *MessageID,
const char *RelatesTo,
const char *EndpointReference,
const char *Types,
const char *Scopes,
const char *MatchBy,
const char *XAddrs,
unsigned int MetadataVersion)

void wsdd_event_Bye(**struct** soap *soap,
unsigned int InstanceId,
const char *SequenceId,
unsigned int MessageNumber,
const char *MessageID,
const char *RelatesTo,
const char *EndpointReference,
const char *Types,
const char *Scopes,
const char *MatchBy,
const char *XAddrs,
unsigned int MetadataVersion)

soap_wsdd_mode wsdd_event_Probe(**struct** soap *soap,
const char *MessageID,
const char *ReplyTo,
const char *Types,
const char *Scopes,
const char *MatchBy,
struct wsdd__ProbeMatchesType *ProbeMatches)

void wsdd_event_ProbeMatches(**struct** soap *soap,
unsigned int InstanceId,
const char *SequenceId,
unsigned int MessageNumber,
const char *MessageID,
const char *RelatesTo,
struct wsdd__ProbeMatchesType *ProbeMatches)

soap_wsdd_mode wsdd_event_Resolve(**struct** soap *soap,
const char *MessageID,
const char *ReplyTo,
const char *EndpointReference,
struct wsdd__ResolveMatchesType *ResolveMatches)

void wsdd_event_ResolveMatches(**struct** soap *soap,
unsigned int InstanceId,

```
        const char *SequenceId,
        unsigned int MessageNumber,
        const char *MessageID,
        const char *RelatesTo,
        const char *EndpointReference,
        const char *Types,
        const char *Scopes,
        const char *MatchBy,
        const char *XAddrs,
        unsigned int MetadataVersion)
```

These event handlers will be invoked when inbound WS-Discovery messages arrive using:

```
if (!soap_valid_socket(soap_bind(soap, NULL, port, 100)))
    .. error
if (soap_wsdd_listen(soap, timeout))
    ... error
```

which will listen for timeout seconds to inbound WS-Discovery messages on a port and dispatches them to the event handlers. A negative timeout is measured in ns.