Contents

1 Introduction 5

2 Notational Conventions 6

3 Interoperability 6

4 Quick User Guide 7

4.1 How to Use the gSOAP Stub and Skeleton Compiler to Build SOAP Clients . . . 8
  4.1.1 Example ................................................................. 8
  4.1.2 Namespace Considerations ........................................... 11
  4.1.3 Example ................................................................. 12
  4.1.4 Some SOAP Encoding Considerations .............................. 13
  4.1.5 Example ................................................................. 14
  4.1.6 How to Change the Response Element Name ...................... 15
  4.1.7 Example ................................................................. 15
  4.1.8 How to Specify Multiple Output Parameters ...................... 16
  4.1.9 Example ................................................................. 16
  4.1.10 How to Specify Output Parameters With Complex Data Types .... 18
  4.1.11 Example ................................................................. 18
  4.1.12 How to Specify Anonymous Parameter Names .................... 20
  4.1.13 How to Specify a Method with No Input Parameters .......... 21

4.2 How to Use the gSOAP Stub and Skeleton Compiler to Build SOAP Web Services 21
  4.2.1 Example ................................................................. 22
  4.2.2 How to Create a Stand Alone Service ............................ 24
  4.2.3 Some Web Service Implementation Issues .......................... 25
  4.2.4 How to Generate WSDL Service Descriptions .................... 25

1
1 Introduction

The gSOAP toolkit provides a unique SOAP-to-C/C++ language binding for the development of SOAP Web Services and clients. Other SOAP C++ implementations adopt a SOAP-centric view and offer SOAP APIs for C++ that require the use of class libraries for SOAP-like data structures. This often forces a user to adapt the application logic to these libraries. In contrast, gSOAP provides a C/C++ transparent SOAP API through the use of compiler technology that hides irrelevant SOAP-specific details from the user. The gSOAP stub and skeleton compiler automatically maps native and user-defined C and C++ data types to semantically equivalent SOAP data types and vice-versa. As a result, full SOAP interoperability is achieved with a simple API relieving the user from the burden of SOAP details and enables him or her to concentrate on the application-essential logic. The compiler enables the integration of (legacy) C/C++ and Fortran codes (through a Fortran-to-C interface), embedded systems, and real-time software in SOAP applications that share computational resources and information with other SOAP applications, possibly across different platforms, language environments, and disparate organizations located behind firewalls.

gSOAP minimizes application adaptation for building SOAP clients and Web Services. The gSOAP compiler generates SOAP marshalling routines that (de)serialize application-specific C/C++ data structures. gSOAP includes a WSDL generator to generate Web service descriptions for your Web services. A WSDL converter for generating client stubs is under development. This converter "closes the circle" in that it enables client development without the need for users to analyze Web service details to implement a client.

Some of the highlights of gSOAP are:

- Unique interoperability features: the gSOAP compiler generates SOAP marshalling routines that (de)serialize native and user-defined C/C++ data structures. gSOAP is also one of the few SOAP toolkits that support the full range of SOAP 1.1 features including multi-dimensional arrays and polymorphic types. For example, a remote method with a base class parameter may accept derived class instances from a client. Derived class instances keep their identity through dynamic binding. To this end, gSOAP supports method overloading and overriding.

- gSOAP includes a WSDL generator for convenient Web Service publishing.

- Automatic stand-alone SOAP client and server code generation. Support for incorporating multiple remote methods in one Web service application.

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

- Platform independent: Windows, Unix, Linux, Pocket PC, etc.

- Fast in situ serialization and deserialization with SOAP encoding of arbitrary user-defined and built-in C and C++ data structures.

- Fully SOAP 1.1 compliant data encoding and decoding. (Also SOAP 1.2 compliant, except for header faults, SOAP actors, SOAP root.)

- The built-in specialized XML pull parser for SOAP is fast and efficient and does not require intermediate data storage for demarshalling which saves space and time.
• 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 (and received) by small-memory devices such as PDAs.

• Supports C++ single class inheritance, dynamic binding, overloading, arbitrary pointer struc-
tures such as lists, trees, graphs, cyclic graphs, fixed-size arrays, (multi-dimensional) dy-
namic arrays, enumerations, built-in XML 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 remote method 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 commands.

Bold font Denotes C and C++ keywords.

Courier font Denotes HTTP header content, HTML, XML, and XML schema fragments.

[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 Interoperability

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

Apache 2.2
4 Quick User Guide

This user guide offers a quick way to get started with gSOAP. This section requires a basic understanding of the SOAP 1.1 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 stub and skeleton compiler without a detailed understanding of the SOAP protocol when the applications are build 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 SOAP C/C++ Web services and clients that connect to other existing SOAP implementations such as Apache SOAP and SOAP::Lite for which some details of the SOAP protocol need to be understood.
### 4.1 How to Use the gSOAP Stub and Skeleton Compiler to Build SOAP Clients

In general, the implementation of a SOAP client application requires a stub routine for each remote method that the client application needs to invoke. The primary stub’s responsibility is to marshall the input data, send the request to the designated SOAP service over the wire, to wait for the response, and to demarshall the output data when it arrives. The client application invokes the stub routine for a remote method as if it would invoke a local method. To write a stub routine in C or C++ by hand is a tedious task, especially if the input and/or output data structures of a remote method are complex data types such as records, arrays, and graphs.

The generation of stub routines for a SOAP client is fully automated with gSOAP. The gSOAP stub and skeleton compiler is a preprocessor that generates the necessary C++ sources to build SOAP C++ clients. The input to the gSOAP stub and skeleton compiler consists of a standard C/C++ header file. The SOAP remote methods are specified in this header file as function prototypes. Stub routines in C/C++ source form are automatically generated by the gSOAP compiler for these function prototypes of remote methods. The stub routines allow C and C++ client applications to seamlessly interact with existing SOAP Web services.

The gSOAP stub and skeleton compiler also generates skeleton routines for each of the remote methods specified in the header file. The skeleton routines can be readily used to implement one or more of the remote methods 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).

The input and output parameters of a SOAP remote method may be simple data types or complex data types. The necessary type declarations of C/C++ user-defined data structures such as structs, classes, enumerations, arrays, and pointer-based data structures (graphs) are to be provided in the header file. The gSOAP stub and skeleton compiler automatically generates serializers and deserializers for the data types to enable the generated stub routines to encode and decode the contents of the parameters of the remote methods.

The remote method name and its parameterization can be found with a SOAP Web service description, typically in the form of an XML schema. There is an almost one-to-one correspondence between the XML schema description of a SOAP remote method and the C++ type declarations required to build a client application for the Web service. The schemas are typically part of the WSDL (Web Service Description Language) specification of a SOAP Web service. A WSDL converter for conversion of WSDL service descriptions into a header file for gSOAP for an automated implementation of client stubs is currently under development.

#### 4.1.1 Example

The `getQuote` remote method of XMethods Delayed Stock Quote service provides a delayed stock quote for a given ticker name, see [http://xmethods.com/detail.html?id=2](http://xmethods.com/detail.html?id=2) for details. The description of the Delayed Stock Quote service provides the following details:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Endpoint URL</td>
<td><a href="http://services.xmethods.net:80/soap">http://services.xmethods.net:80/soap</a></td>
</tr>
<tr>
<td>SOAP action</td>
<td>&quot;&quot; (2 quotes)</td>
</tr>
<tr>
<td>Remote method namespace</td>
<td>urn:xmethods-delayed-quotes</td>
</tr>
<tr>
<td>Remote method name:</td>
<td>getQuote</td>
</tr>
<tr>
<td>Input parameter:</td>
<td>symbol of type xsd:string</td>
</tr>
<tr>
<td>Output parameter:</td>
<td>Result of type xsd:float</td>
</tr>
</tbody>
</table>
The following `getQuote.h` header file declares the remote method in C++:

```cpp
// Content of file "getQuote.h":
int ns1__getQuote(char *symbol, float &Result);
```

The remote method is declared as a `ns1__getQuote` function prototype which specifies all of the necessary details for the gSOAP compiler to generate the stub routine for a client application to interact with the Delayed Stock Quote service.

The Delayed Stock Quote service description requires that the input parameter of the `getQuote` remote method is a `symbol` parameter of type string. The description also indicates that the `Result` output parameter is a floating point number that represents the current unit price of the stock in dollars. The gSOAP compiler uses the convention the last parameter of the function prototype must be the output parameter of the remote method, which is required to be passed by reference using the reference operator (`&`) or by using a pointer type. All other parameters except the last are input parameters of the remote method, which are required to be passed by value or passed using a pointer to a value (by reference is not allowed). The function prototype associated with a remote method is required to return an `int`, whose value indicates to the caller whether the connection to a SOAP Web service was successful or resulted in an exception, see Section 6.2 for the error codes.

The use of the namespace prefix `ns1__` in the remote method name in the function prototype declaration is discussed in detail in 4.1.2. Basically, a namespace prefix is distinguished by a pair of underscores in the function name, as in `ns1__getQuote` where `ns1` is the namespace prefix and `getQuote` is the remote method name. (A single underscore in an identifier name will be translated to a dash in the XML output, see Section 6.3.)

The gSOAP compiler is invoked from the command line with:

```
soapcpp getQuote.h
```

The compiler generates the stub routine for the `getQuote` remote method specified in the `getQuote.h` header file. This stub routine can be called by a client program at any time to request a stock quote from the Delayed Stock Quote service. The interface to the generated stub routine is called a proxy, which is the following function generated by the gSOAP compiler:

```cpp
int soap_call_ns1__getQuote(char *URL, char *action, char *symbol, float &Result);
```

The stub routine of the proxy is saved in `soapClient.cpp`. The file `soapC.cpp` contains the serializer and deserializer routines for the data types used by the stub.

Note that the parameters of the generated `soap_call_ns1__getQuote` proxy are identical to the `ns1__getQuote` function prototype with two additional input parameters: `URL` is the SOAP Web service endpoint `URL` passed as a string and `action` is a string that denotes the SOAP action required by the Web service.

The following example C++ client program invokes the proxy to retrieve the latest AOL stock quote from the XMethods Delayed Stock Quote service:

```cpp
#include "soapH.h" // obtain the generated proxy
main()
{
   float quote;
```
```cpp
if (soap_call_ns1_getQuote("http://services.xmethods.net:80/soap", ",", "AOL", quote)
== SOAP_OK)
    cout << "Current AOL Stock Quote = " << quote;
else // an error occurred
    soap_print_fault(stderr); // display the SOAP fault message on the stderr stream
}
```

The XMethods Delayed Stock Quote service endpoint URL is `http://services.xmethods.net/soap` port 80 and the SOAP action required is "" (two quotes). If successful, the proxy returns SOAP_OK and quote contains the latest stock quote. Otherwise, an error occurred and the SOAP fault is displayed with the soap_print_fault function.

Compare this simple C++ client application to an implementation in Java using Apache SOAP, see [http://www.xmethods.net/download/servicefiles/stockquoteClient.java](http://www.xmethods.net/download/servicefiles/stockquoteClient.java).

When the example client application is invoked, the SOAP request is performed by the proxy routine `soap_call_ns1_getQuote`, which generates the following SOAP request message:

```
POST /soap HTTP/1.1
Host: services.xmethods.net
Content-Type: text/xml
Content-Length: 529
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema"
    xmlns:ns1="urn:xmethods-delayed-quotes"
    SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <SOAP-ENV:Body>
    <ns1:getQuote>
    <symbol>AOL</symbol>
    </ns1:getQuote>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The XMethods Delayed Stock Quote service responds with the SOAP response message:

```
HTTP/1.1 200 OK
Date: Sat, 25 Aug 2001 19:28:59 GMT
Content-Type: text/xml
Server: Electric/1.0
Connection: Keep-Alive
Content-Length: 491

<?xml version='1.0' encoding='UTF-8'?>
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/
    xmlns:xsi='http://www.w3.org/1999/XMLSchema-instance'
    xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
    soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
    </soap:Body>
</soap:Envelope>
```
<soap:Body>
<n:getQuoteResponse xmlns:n='urn:xmethods-delayed-quotes'>
<Result xsi:type='xsd:float'>41.81</Result>
</n:getQuoteResponse>
</soap:Body>
</soap:Envelope>

The server’s response is parsed by the stub routine of the proxy of the client. The stub routine further demarshalls the data of Result element of the SOAP response and stores it in the quote parameter of soap.call.ns1.getQuote.

A client program can invoke a remote method at any time and multiple times if necessary. Consider for example:

```c
...  
float quotes[3]; char *myportfolio[] = {"IBM", "AOL", "MSDN"};
for (int i = 0; i < 3; i++)
  if (soap.call.ns1.getQuote("http://services.xmethods.net:80/soap", ",", myportfolio[i], quotes[i]) != SOAP_OK)
    break;
  if (soap.error) // an error occurred
    soap.print_fault(stderr);
...```

This client composes an array of stock quotes by calling the ns1.getQuote proxy for each symbol in a portfolio array.

This example demonstrated how easy it is to build a SOAP client with gSOAP once the details of a Web service are available and understood.

4.1.2 Namespace Considerations

The declaration of the ns1.getQuote function prototype (discussed in the previous section) uses the namespace prefix ns1 of the remote method namespace, which is distinguished by a pair of underscores in the function name to separate the namespace prefix from the remote method name. The purpose of a namespace prefix is to associate a remote method name with a service in order to prevent naming conflicts, e.g. to distinguish identical remote method names used by different services.

Note that the XML response of the XMethods Delayed Stock Quote service example uses the namespace prefix n which is associated with the namespace URI urn:xmethods-delayed-quotes through the xmlns:n=urn:xmethods-delayed-quotes binding. The use of namespace prefixes and namespace URIs is also required to enable SOAP applications to validate the content of a client’s request and vice versa. The namespace URI 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 the client program. 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 stub and skeleton compiler.

The namespace mapping table for the Delayed Stock Quote client is:
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 `ns1`, which is bound to `urn:xmethods-delayed-quotes` by the namespace mapping table shown above, is used by the generated stub routine to encode the `getQuote` request. This is performed automatically by the gSOAP compiler by using the `ns1` prefix of the `ns1__getQuote` method name specified in the `getQuote.h` header file. In general, if a function name of a remote method, `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:

```xml
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:ns1="urn:xmethods-delayed-quotes"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

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

### 4.1.3 Example

The incorporation of namespace prefixes into C++ identifier names is necessary to distinguish remote methods that share the same name but are implemented in different Web services. Consider for example:

```c++
// Contents of file "getQuote.h":
int ns1__getQuote(char *symbol, float &Result);
int ns2__getQuote(char *ticker, char &quote);
```

The namespace prefix is separated from the remote method names by a pair of underscores (__) by convention.

This example enables a client program to connect to a (hypothetical) Stock Quote service with remote methods that can only be distinguished by their namespaces. Consequently, two different namespace prefixes have been used as part of the remote method names.
The namespace prefix convention can also be applied to class declarations that contain SOAP compound values that share the same name but have different namespaces that refer to different XML schemas. For example:

```c
class e_Address // an electronic address
{
  char *email;
  char *url;
};

class s_Address // a street address
{
  char *street;
  int number;
  char *city;
};
```

The namespace prefix is separated from the data type names by a pair of underscores (\_\_) by convention.

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

```xml
<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:

```xml
<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:

```c
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.

### 4.1.4 Some SOAP 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 stub and skeleton compiler, assumes SOAP
encoding. This can be easily changed by using **typedef** definitions in the header file input to the gSOAP compiler. The type name defined by a **typedef** definition corresponds to an XML schema type and may include an optional namespace prefix. For example, the following **typedef** declarations, when part of the header file input to the gSOAP compiler, defines various built-in XML schema types implemented as primitive C/C++ types:

```c
// 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 simple mechanism informs the gSOAP compiler to generate serializers and deserializers that explicitly encode and decode the primitive C++ types as built-in primitive XML schema types. At the same time, the use of **typedef** as a means to inform the compiler 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 7.2.2 for alternative class implementations of primitive XML schema types which allows for the marshalling of polymorphic primitive SOAP types).

### 4.1.5 Example

Reconsider the *getQuote* example, now rewritten with explicit XML schema types to illustrate the effect:

```c
// Contents of file "getQuote.h":
typedef char *xsd_string;
typedef float xsd_float;
int ns1_getQuote(xsd_string symbol, xsd_float &Result);
```

This header file is compiled by the gSOAP stub and skeleton compiler and the compiler generates source code for the function `soap_call_ns1_getQuote`, which is identical to the “old” proxy (without the **typedefs**):

```c
int soap_call_ns1_getQuote(char *URL, char *action, char *symbol, float &Result);
```

The client application does not need to be rewritten and can still call the proxy with the “old” parameter types. 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 schema types.

For example, when the client application calls the proxy, the proxy produces a SOAP request with `xsd:string`:

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

14
The service response is:

```xml
<soap:Body>
  <n:getQuoteResponse xmlns:n='urn:xmethods-delayed-quotes'>
    <Result xsi:type='xsd:float'>41.81</Result>
  </n:getQuoteResponse>
</soap:Body>
```

The validation of this service response by the stub routine takes place by matching the namespace URIs that are bound to the xsd namespace prefix. The stub also expects the getQuoteResponse element to be associated with namespace URI urn:xmethods-delayed-quotes through the binding of the namespace prefix ns1 in the namespace mapping table. The service response uses namespace prefix n for the getQuoteResponse element. This namespace prefix is bound to the same namespace URI urn:xmethods-delayed-quotes and therefore the service response is assumed to be valid. The response is rejected and a SOAP fault is generated if the namespace URIs do not match.

### 4.1.6 How to Change the Response Element Name

There is no explicit standard convention for the response element name in SOAP, although it is recommended that the response element name is the method name ending with "Response". For example, the response element of getQuote is getQuoteResponse.

The response element name can be specified explicitly using a struct or class declaration in the header file. The struct or class name represents the SOAP response element name used by the service. Consequently, the output parameter of the remote method 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 gSOAP compiler to automatically generate a struct for the response which is internally used by a stub.

### 4.1.7 Example

Reconsider the getQuote remote method specification which can be rewritten with an explicit declaration of a SOAP response element as follows:

```c
// Contents of "getQuote.h":
typedef char *xsd_string;
typedef float xsd_float;
struct ns1_getQuoteResponse {xsd_float Result;};
int ns1_getQuote(xsd_string symbol, struct ns1_getQuoteResponse &r);
```

The SOAP request is the same as before:
The difference is that the service response is required to match the specified getQuoteResponse name and its namespace URI:

```
...<soap:Body>
<n:getQuoteResponse xmlns:n='urn:xmethods-delayed-quotes'>
<Result xsi:type='xsd:float'>41.81</Result>
</n:getQuoteResponse>
</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.

Note that the `struct` (or `class`) declaration may appear within the function prototype declaration. For example:

```
    // Contents of "getQuote.h":
    typedef char *xsd_string;
    typedef float xsd_float;
    int ns1_getQuote(xsd_string symbol, struct ns1_getQuoteResponse {xsd_float Result;}&r);
```

This example combines the declaration of the response element of the remote method with the function prototype of the remote method.

### 4.1.8 How to Specify Multiple Output Parameters

The gSOAP stub and skeleton compiler uses the convention that the single output parameter of a remote method is the last parameter of the function prototype declaration in a header file. All other parameters are considered input parameters of the remote method. To specify a remote method with multiple output parameters, a `struct` or `class` must be declared for the remote method response, see also 4.1.6. The fields of the `struct` or `class` are the output parameters of the remote method. 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 4.1.12.

### 4.1.9 Example

As an example, consider a hypothetical remote method `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);
```

16
The gSOAP stub and skeleton compiler 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 xsi:type="ns3:getNames">  
  <SSN xsi:type="xsd:string">999 99 9999</SSN>  
  </ns3:getNames>  
  ...
```

The response by a SOAP service could be:

```
...  
  <m:getNamesResponse xmlns:m="urn:names">  
  <first xsi:type="xsd:string">John</first>  
  <last xsi:type="xsd:string">Doe</last>  
  </m:getNamesResponse>  
  ...
```

where first and last are the output parameters of the getNames remote method of the service.

As another example, consider a remote method 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 remote method response enables the declaration of remote methods that have parameters that are passed both as input and output parameters.

The gSOAP 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 xsi:type="X-rox:copy-name">  
  <name xsi:type="xsd:string">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 xsi:type="xsd:string">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.
4.1.10 How to Specify Output Parameters With Complex Data Types

If the output parameter of a remote method is a complex data type such as a `struct` or `class` it is necessary to specify the response element of the remote method 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 4.1.6.

4.1.11 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, see http://xmethods.com/detail.html?id=86 for details. The remote method 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 6.3 for details on the translation of C++ identifiers to XML element names.

The SOAP C++ stub and skeleton 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:

```
...  soap_call_ns1_getFlightInfo("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/1999/XMLSchema-instance"},
    {"xsd", "http://www.w3.org/1999/XMLSchema"},
    {"ns1", "urn:galdemo:flighttracker"},
```
When invoked by a client application, the proxy produces the SOAP request:

```xml
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"?>
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/
 xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/1999/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:

```xml
HTTP/1.1 200 ok
Date: Thu, 30 Aug 2001 00:34:17 GMT
Server: IBM_HTTP_Servlet/1.3.12.3 Apache/1.3.12 (Win32)
Set-Cookie: sesessionid=2GFVT0GC3DOLRGU2L4HFA;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'?>
 xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/1999/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>
  </return>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```
The proxy returns the service response in variable $r$ of type \textbf{struct} \texttt{ns1\_getFlightInfoResponse} and this information can be displayed by the client application with the following code fragment:

```cpp
    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.

4.1.12 \textbf{How to Specify Anonymous Parameter Names}

The SOAP 1.1 protocol allows parameter names to be anonymous. That is, the name(s) of the output parameters of a remote method are not strictly required to match a client’s view of the parameters names. Also, the input parameter names of a remote method are not strictly required to match a service’s view of the parameter names. Although this convention is likely to be deprecated, the gSOAP compiler can generate stub and skeleton routines that support anonymous parameters.

To make parameter names 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:

```cpp
    // Contents of "getQuote.h":
    typedef char *xsd\_string;
    typedef float xsd\_float;
    int ns1\_getQuote(xsd\_string symbol, &r);
```

Or, alternatively with a response \textbf{struct}:

```cpp
    // Contents of "getQuote.h":
    typedef char *xsd\_string;
    typedef float xsd\_float;
    struct ns1\_getQuoteResponse {xsd\_float \_return;};
    int ns1\_getQuote(xsd\_string symbol, struct ns1\_getQuoteResponse &r);
```

In this example, \_return is an anonymous output parameter. 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.

\textbf{Caution}: when anonymous parameter names are used, the order of the parameters in the function prototype of a remote method is significant.
### 4.1.13 How to Specify a Method with No Input Parameters

To specify a remote method that has no input parameters, just provide a function prototype with one parameter which is the output parameter. However, some C/C++ compilers (notably Visual C++™) will not compile and complain about an empty `struct`. 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:

```c
struct ns3_SoapService
{
    public:
    int id;
    char *name;
    char *owner;
    char *description;
    char *homepageURL;
    char *endpoint;
    char *soapAction;
    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. To call the proxy, use NULL as the actual input parameter value. Most C/C++ compilers allow empty `structs` and therefore the `void*` parameter is not required.

### 4.2 How to Use the gSOAP Stub and Skeleton Compiler to Build SOAP Web Services

The gSOAP stub and skeleton compiler generates skeleton routines in C++ source form for each of the remote methods specified as function prototypes in the header file processed by the gSOAP compiler. The skeleton routines can be readily used to implement the remote methods in a new SOAP Web service. The compound data types used by the input and output parameters of SOAP remote methods 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 remote method. The gSOAP 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 remote methods. The gSOAP compiler also generates a remote method 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.
4.2.1 Example

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

```c
// 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 compiler as the `xsd:double` XML 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 compiler is invoked from the command line with:

```
soapcpp calc.h
```

The compiler generates the skeleton routines for the `add`, `sub`, and `sqrt` remote methods 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 remote method requests to the appropriate skeletons to serve the requests. The skeleton in turn calls the remote method implementation function. The function prototype of the remote method implementation function is specified in the header file that is input to the gSOAP compiler.

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

```c
// Contents of file "calc.cpp":
#include "soapH.h"
#include <math.h> // for sqrt()
main()
{
    soap__serve(); // use the remote method request dispatcher
}
// Implementation of the "add" remote method:
int ns__add(double a, double b, double &result)
{
    result = a + b;
    return SOAP_OK;
}
// Implementation of the "sub" remote method:
int ns__sub(double a, double b, double &result)
{
    result = a - b;
}
```
The implementation of the remote methods 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. The exception will be passed on to the client.

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

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 remote method.)

The header file input to the gSOAP 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 proxy is available
from the soapClient.cpp file after invoking the gSOAP 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.

4.2.2 How to Create a Stand Alone Service

The deployment of a Web service as a CGI application is an easy means to provide your service
on the Internet. Services can also run as stand-alone services on intranets where client-service
interactions are not blocked by firewalls.

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

```c
main()
{
    int m, s;
    m = soap_bind("machine.cs.fsu.edu", 18083, 100);
    if (m ¡ 0)
    {
        soap_print_fault(stderr);
        exit(-1);
    }
    fprintf(stderr, "Socket connection successful: master socket = %d\n", m);
    for (int i = 1; ; i++)
    {
        s = soap_accept();
        if (s < 0)
        {
            soap_print_fault(stderr);
            exit(-1);
        }
        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);
        soap_server(); // process RPC skeletons
        fprintf(stderr, "request served\n");
        soap_end(); // clean up everything and close socket
    }
}
```

The functions used are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>soap_bind(host, port, backlog)</td>
<td>Returns master socket (backlog = max. queue size for requests)</td>
</tr>
<tr>
<td>soap_accept()</td>
<td>Returns slave socket</td>
</tr>
<tr>
<td>soap_end()</td>
<td>Clean up deserialized data (except class instances) and temporary data</td>
</tr>
<tr>
<td>soap_free()</td>
<td>Clean up temporary data only</td>
</tr>
<tr>
<td>soap_destroy()</td>
<td>Clean up deserialized class instances</td>
</tr>
</tbody>
</table>
The *host* name in `soap_bind` may be NULL to indicate that the current host should be used. The `soap_accept_timeout` global variable specifies the timeout value for `soap_accept()`. See Section 5.7 for more details on memory management.

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

### 4.2.3 Some Web Service Implementation Issues

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

- 4.1.2 for namespace considerations.
- 4.1.4 for an explanation on how to change the encoding of the primitive types.
- 4.1.6 for a discussion on how the response element format can be controlled.
- 4.1.8 for details on how to pass multiple output parameters from a remote method.
- 4.1.10 for passing complex data types as output parameters.
- 4.1.12 for anonymizing the input and output parameter names.

### 4.2.4 How to Generate WSDL Service Descriptions

The gSOAP stub and skeleton compiler `soapcpp` generates WSDL (Web Service Description Language) service descriptions and XML schema files when processing a header file. The compiler produces one WSDL file for a set of remote methods. The names of the function prototypes of the remote methods 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 remote methods, multiple WSDL files will be created and each file describes the set of remote methods belonging to a namespace prefix.

To publish the WSDL service description, the `%{}`-patterns that appear in the generated WSDL file have to be filled in with the following information:

<table>
<thead>
<tr>
<th>Replace</th>
<th>With</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%{Service}%</code></td>
<td>the file name of the CGI service application (without a file name extension)</td>
</tr>
<tr>
<td><code>%{URL}%</code></td>
<td>the endpoint URL of the service (without the CGI file name)</td>
</tr>
<tr>
<td><code>%{URI}%</code></td>
<td>the namespace URI of the service (can be the same as the URL)</td>
</tr>
</tbody>
</table>

This information can also be provided in the header file which will then be automatically incorporated in the WSDL file and the namespace mapping table, see advanced features Section 10.1.

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:

```c
struct Namespace namespaces[] =
{
    {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
```
After replacing the %{}% patterns in the namespace mapping table file, this file can be incorporated in the client/service application, see Section 6.4 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 file permissions are set right (chmod 755 calc.cgi for Unix/Linux). You can then publish the WSDL file on the Web.

The gSOAP compiler also generates XML schema files for all C/C++ complex types (e.g. structs and classes) 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 types.

4.2.5 Example

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

```c
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 remote methods:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="%{Service}%"
xmlns="http://schemas.xmlsoap.org/wsdl/
targetNamespace="%{URL}%/ns.xsd"
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="%{URL}%/ns.xsd"
xmlns:ns="%{URL}%/ns.xsd">
<types>
<schema
xmlns="http://www.w3.org/2000/10/XMLSchema"
targetNamespace="%{URL}%/ns.xsd"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/
<complexType name="addResponse">
```

26
<element name="result" type="double" minOccurs="0" maxOccurs="1"/>
</all>
<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="%(Service)%PortType">
  <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="%(Service)%Binding" type="tns:%(Service)%PortType"/>
The service name of the calculator service could be `calc` (so the file name of the CGI service application is `calc.cgi`), the URL could be `http://www.mycalc.com`, and the namespace URI could be `http://www.mycalc.com` (the URI must be unique, if possible, and the URL uniquely identifies an organization). According to this, the following modifications to the generated WSDL file have to be made:

<table>
<thead>
<tr>
<th>Replace</th>
<th>With</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%{Service}%</code></td>
<td><code>calc</code></td>
</tr>
<tr>
<td><code>%{URL}%</code></td>
<td><code>http://www.mycalc.com</code></td>
</tr>
<tr>
<td><code>%{URI}%</code></td>
<td><code>http://www.mycalc.com</code></td>
</tr>
</tbody>
</table>
4.2.6 Combining a Client and Service into a Peer Application

This 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.

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 compiler is given below:

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

The quotex.cpp client/service application source is:

```c
// Contents of file "quotex.cpp":
#include "soapH.h" // include generated proxy and SOAP support
int main(int argc, char **argv)
{
    float q;
    if (argc != 2)
        soap__serve();
    else if (soap__call_ns3__getQuote("http://www.cs.fsu.edu/~engelen/quotex.cgi", NULL, argv[1], argv[2], q))
        soap__print_fault(stderr);
    else
        printf("Company %s: %f (%s)n", argv[1], q, argv[2]);
    return 0;
}
int ns3__getQuote(char *symbol, char *country, float &result)
{
    float q, r;
    if (soap__call_ns1__getQuote("http://services.xmethods.net/soap", NULL, symbol, q) == 0 &&
        soap__call_ns2__getRate("http://services.xmethods.net/soap", NULL, "us", country, r) == 0)
    {
        result = q*r;
        return SOAP_OK;
    }
    else
        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 remote methods. 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(char *symbol, float &result) 
* { return SOAP_NO_METHOD; } // dummy: will never be called 
* int ns2__getRate(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/"}, 
*     {"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:
* 
* soapcpp quotex.h 
* g++ -o quotex.cgi quotex.cpp soapC.cpp soapClient.cpp soapServer.cpp stdsoap.cpp -lsocket -lxnet 
* -lnsl -lm 
* 
* Note: under Linux you can 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 AOL uk 
* 
* returns the quote of AOL 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 remote methods, 
* while the client part will only use the stubs and the service part will use the skeletons. Thus, 
* dummy implementations of the unused remote methods 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).
4.3 How to Use gSOAP for Asynchronous SOAP Messaging

The default gSOAP client-server interaction is synchronous: the client waits for the server to respond to the request. gSOAP also supports asynchronous SOAP messaging. SOAP messaging routines are declared as function prototypes, just like remote methods for SOAP RPC. However, the output parameter is a \texttt{void} type to indicate the absence of a return value.

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

\begin{verbatim}
int ns_event(int eventNo, void dummy);
\end{verbatim}

The gSOAP stub and skeleton compiler generates the following functions in \texttt{soapClient.cpp}:

\begin{verbatim}
int soap_send_ns_event(const char URL, const char action, int event);
int soap_recv_ns_event(struct ns_event *dummy);
\end{verbatim}

The \texttt{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 \texttt{soap_closesock()}. The open socket connection can also be used to obtain a service response, e.g. with a \texttt{soap_recv} function call.

The \texttt{soap_recv_ns_event} function waits for a SOAP message on the currently open socket (\texttt{soap_socket}) and fills the \texttt{struct ns_event} with the \texttt{ns_event} parameters (e.g. \texttt{int eventNo}). The \texttt{struct ns_event} is automatically created by gSOAP and is a mirror image of the \texttt{ns_event} parameters:

\begin{verbatim}
struct ns_event
{
  int eventNo;
}
\end{verbatim}

The gSOAP generated \texttt{soapServer.cpp} code includes a skeleton routine to accept the asynchronous message. (The skeleton routine does not respond with a SOAP response message.)

\begin{verbatim}
int soap_server_ns_event();
\end{verbatim}

The skeleton routine calls the user-implemented \texttt{ns_event(int eventNo)} routine (note the absence of the \texttt{void} parameter!).

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

\begin{verbatim}
main()
{
  soap_server();
}
int ns_event(int eventNo)
{
  ... /* handle event
  return SOAP_OK;
}
\end{verbatim}
4.4 How to Separately Use the SOAP Serializers and Deserializers

The gSOAP stub and skeleton compiler generates serializers and deserializers for all 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 data in XML format that is possibly saved by other applications.

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>soap_socket</td>
<td>socket file descriptor for input and output or -1</td>
</tr>
<tr>
<td>soap_sendfd</td>
<td>if soap_socket&lt;0, file descriptor for send operations</td>
</tr>
<tr>
<td>soap_recvfd</td>
<td>if soap_socket&lt;0, file descriptor for receive operations</td>
</tr>
<tr>
<td>soap_buffering</td>
<td>when not zero, a send buffer is used</td>
</tr>
</tbody>
</table>

The soap_sendfd and soap_recvfd file descriptors will only be used when soap_socket is not in use (i.e. soap_socket<0).

The following initializing and finalizing functions can be used:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void soap_begin_send()</td>
<td>use buffered socket sends when soap_socket≥0</td>
</tr>
<tr>
<td>int soap_end_send()</td>
<td>flush the buffer</td>
</tr>
<tr>
<td>int soap_begin_recv()</td>
<td>if an HTTP header is present, parse it first</td>
</tr>
<tr>
<td>int soap_end_recv()</td>
<td>perform a id/href consistancy check on deserialized data</td>
</tr>
</tbody>
</table>

4.4.1 Serializing a Data Type

To serialize a data type, two functions need to be called to process the data. The first function (soap_serialize) analyzes pointers and determines if multi-references are required to encode the data and if the data contains cycles. The second function (soap_put) generates the SOAP encoding output for that data type.

The function names are specific to a data type. For example, soap_serialize_float(&d) is called to serialize an float value and soap_put_float(&d, "number", NULL) is called to output the floating point value in SOAP tagged with the name <number>. To initialize data, the soap_default function of a data type can be used. For example, soap_default_float(&d) initializes the float to 0.0. The soap_default functions are useful to initialize complex data types such as arrays, structs, and class instances.

The following table lists the type naming conventions used:
Consider for example the following declaration of `p` as a pointer to a `struct ns::Person`:

```c
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 compiler:

```c
soap_serialize_PointerTons::Person(&p);
```

The address of `p` is passed, so the serializer can determine whether `p` was already serialized and to discover cycles in graph data structures. 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`):

```c
soap_put_PointerTons::Person(&p, "ns:element-name", "ns:type-name");
```

This produces:

```xml
<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` function. All temporary data structures and data structures deserialized on the heap are destroyed with the `soap_end()` function. The `soap_free()` function can be used to remove the temporary data only and keep the deserialized data on the
heap. 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.

If more than one data structure is to be serialized and parts of those data structures are shared through pointers, then the soap serialize functions MUST to be called first before any of the soap put functions. This is necessary to ensure that multi-reference data shared by the data structures is encoded as multi-reference.

For example, to encode the contents of two variables var1 and var2 the serializers are called before the output routines:

```c
T1 var1;
T2 var2;
...
soap_begin(); // start new serialization phase
soap_enable_embedding = 1; // do not use independent elements
soap_serialize_Type1(&var1);
soap_serialize_Type2(&var2);
...
[soap_socket = a_socket_file_descriptor;]
[soap_sendfd = an_output_file_descriptor;]
[soap_begin_send();] // use buffered socket output
soap_put_Type1(&var1, "[namespace-prefix:]element-name1", "[namespace-prefix:]type-name1");
soap_put_Type2(&var2, "[namespace-prefix:]element-name2", "[namespace-prefix:]type-name2");
...
[soap_end_send();] // flush buffered socket output
soap_end(); // remove temporary data structures
...
```

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. The output stream is set by the assignment to the soap_sendfd variable.

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

In principle, encoding MAY take place without calling the soap serialize functions. However, as the following example demonstrates the resulting encoding is not SOAP 1.1 compliant. However, the messages can still be used with gSOAP to save and restore data.

Consider the following struct:

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

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>

which uses an independent element at the end to represent the multi-referenced integer.

To preserve the exact structure of the data, use the setting soap_enable_embedding=1 (see Section 5.6) to serialize multi-referenced data embedded in the structure which assures the preservation of structure but is not SOAP 1.1 compliant. For example, the resulting 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).

4.4.2 Deserializing a Data Type

To deserialize a data type, its soap_get function is used. The outline of a program that deserializes two variables var1 and var2 is for example:

T1 var1;
T2 var2;
...
soap_begin(); // begin new decoding phase
[soap_recvfd = an_input_stream;]
[soap_begin_recv();] // if HTTP header is present, parse it
soap_get_Type1(&var1, "[namespace-prefix:]element-name1", "[namespace-prefix:]type-name1");
soap_get_Type2(&var2, "[namespace-prefix:]element-name2", "[namespace-prefix:]type-name2");
...
[soap_end_recv();] // check consistancy of id/hrefs
soap_end(); // remove temporary data, including the decoded data on the heap

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. var.soap_get("...", "...").

The soap_begin() call initializes the deserializer. The soap_end() call removes the temporary data structures and the decoded data that was placed on the heap. Temporary data is created only if
the SOAP content includes id and href attributes. An internal hash table is used by the deserializer to bound the id with the href names to reconstruct the shape of the data structure.

To remove temporary data while retaining the deserialized data on the heap, the function soap_free() should be called instead of soap_end().

4.4.3 Example

As an example, consider the following data type declarations:

```c
// 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 store a person named "John" living at Dowling st. 10 in London. He has a mother "Mary" and a father "Stuart". After initialization, the class instance for "John" is serialized and encoded in SOAP to the standard output stream:

```c
// Contents of file "person.cpp":
#include "soapH.h"
main()
{
   ns_Person mother, father, john;
   soap_enable_embedding = 1; // see 5.6
   mother.name = "Mary";
   mother.gender = female;
   mother.address.street = "Dowling 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_begin();
john.soap_serialize();
john.soap_put("johnnie", NULL);
soap_end();
}

struct Namespace namespaces[] =
{
    {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
    {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
    {"xsi", "http://www.w3.org/1999/XMLSchema-instance"},
    {"xsd", "http://www.w3.org/1999/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:

soapcpp person.h

g++ -o person person.cpp soapC.cpp stdsoap.cpp -lsocket -lxnet -lnsl -lm

(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.)

Running the person application results in the SOAP output:

xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/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>

37
The following program fragment decodes this content and reconstructs the original data structure on the heap:

```c
#include "soapH.h"
main()
{
  ns::Person *mother, *father, *john = NULL;
  soap_begin();
  soap_get_ns::Person(john, "johnnie", NULL);
  mother = john->mother;
  father = john->father;
  ...
  soap_free(); // 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/1999/XMLSchema-instance"},
  {"xsd", "http://www.w3.org/1999/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. The following example explicitly passes NULL:

```c
...
  john = soap_get_ns::Person(NULL, "johnnie", NULL);
  ...
```

Note: the second NULL parameter indicates that the schema type attribute of the receiving message can be ignored. The deserializer stores the SOAP contents on the heap, and returns the address. The allocated storage is released with the `soap_end()` call, which removes all temporary and deserialized data from the heap, or with the `soap_free()` call, which removes all temporary data only.

Alternatively, the SOAP content can be decoded within an existing allocated data structure. The following program fragment decodes the SOAP content in a `struct ns::Person` allocated on the stack:

```c
38"
#include "soapH.h"

main()
{
    ns_Person *mother, *father, john;
    soap_begin();
    soap_default_ns_Person(&john);
    soap_get_ns_Person(&john, "johnnie", NULL);
    mother = john->mother;
    father = john->father;
    ...
    soap_free();
}

struct Namespace namespaces[] =
...

Note the use of soap_default_ns_Person. This routine is generated by the gSOAP stub and skeleton compiler and assigns default values to the fields of john.

4.4.4 Default Values for Deserializing Omitted Data

The gSOAP 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 stdsoap.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

When data such as accessors of complex types are omitted in the SOAP payload, the struct and class fields contain default values.

Instead of adding these to stdsoap.h, you can also compile with option -DWITH_USERDEFS_H and include your definitions in file userdefs.h.

5 Using the gSOAP Stub and Skeleton Compiler

The gSOAP stub and skeleton compiler 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:

    soapcpp [ahheaderfile.h]
where aheaderfile.h is a standard C++ header file. The compiler acts as a preprocessor and produces C++ source files that can be used to build SOAP client and Web service applications in C++. The files generated by the compiler are:

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>soapH.h</td>
<td>Main header file to be included by all client and service sources</td>
</tr>
<tr>
<td>soapC.cpp</td>
<td>Serializers and deserializers for the specified data structures</td>
</tr>
<tr>
<td>soapClient.cpp</td>
<td>Client stub routines and proxies for all remote methods</td>
</tr>
<tr>
<td>soapServer.cpp</td>
<td>Service skeleton routines</td>
</tr>
<tr>
<td>soapStub.h</td>
<td>A modified header file produced from the compiler input header file</td>
</tr>
<tr>
<td>.xsd</td>
<td>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 4.2.4</td>
</tr>
<tr>
<td>.wsdl</td>
<td>A ns.wsdl file is generated with an WSDL description for each namespace prefix ns used by a remote method in the header file input to the compiler, see Section 4.2.4</td>
</tr>
<tr>
<td>.nsmap</td>
<td>A ns.nsmap file is generated for each namespace prefix ns used by a remote method in the header file input to the compiler, see Section 4.2.4. The file contains a namespace mapping table that can be used in the client/service sources</td>
</tr>
</tbody>
</table>

Both client and service applications are developed from a header file that specifies the remote methods. 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 ot 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 following files are part of the gSOAP package and are required to build client and service applications:

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdsoap.h</td>
<td>Header file of stdsoap.cpp runtime library</td>
</tr>
<tr>
<td>stdsoap.c</td>
<td>Runtime C library with XML parser and run-time support routines</td>
</tr>
<tr>
<td>stdsoap.cpp</td>
<td>Runtime C++ library identical to stdsoap.c</td>
</tr>
</tbody>
</table>

5.1 Compiler Options

The compiler supports the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h</td>
<td>Print a brief usage message</td>
</tr>
<tr>
<td>-c</td>
<td>Save files using extension .c instead of .cpp</td>
</tr>
<tr>
<td>-d &lt;path&gt;</td>
<td>Save sources in directory specified by &lt;path&gt;</td>
</tr>
<tr>
<td>-p &lt;name&gt;</td>
<td>Save sources with file name prefix &lt;name&gt; instead of “soap”</td>
</tr>
</tbody>
</table>

For example

```bash
soapcpp -cd '../projects' -pmy file.h
```

Saves the sources:

```bash
../projects/myH.h
../projects/myC.c
```
5.2 Compiling a SOAP C++ Client

After invoking the gSOAP stub and skeleton compiler on a header file description of a service, the client application can be compiled on a Linux machine as follows:

```
g++ -o myclient myclient.cpp stdsoap.cpp soapC.cpp soapClient.cpp
```

Or on a Unix machine:

```
g++ -o myclient myclient.cpp stdsoap.cpp soapC.cpp soapClient.cpp -lsocket -lxnet -lm
```

(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:

```c
// Contents of file "myclient.cpp"
#include "soapH.h";
...
// A remote method 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/1999/XMLSchema-instance"},
   {"xsd", "http://www.w3.org/1999/XMLSchema"},
   {"ns1", "urn:my-remote-method"},
   {NULL, NULL}
};
...
```

A mapping table is generated by the gSOAP compiler that can be used in the source, see Section 4.2.4.

5.3 Compiling a SOAP C++ Web Service

After invoking the gSOAP stub and skeleton compiler on a header file description of the service, the server application can be compiled on a Linux machine as follows:
g++ -o myserver myserver.cpp stdsoap.cpp soapC.cpp soapServer.cpp

Or on a Unix machine:

g++ -o myserver myserver.cpp stdsoap.cpp soapC.cpp soapServer.cpp -lsocket -lxnet -lm

(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:

```c
#include "soapH.h"
main()
{
  soap_serve();
}

// Implementations of the remote methods 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/1999/XMLSchema-instance"},
  {"xsd", "http://www.w3.org/1999/XMLSchema"},
  {"ns1", "urn:my-remote-method"},
  {NULL, NULL}
};
```

When the service application is compiled 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 sent to standard output. The method must be implemented in the server application and the type signature of the method must be identical to the remote method 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++ function.

### 5.4 Using gSOAP for Creating Web Services and Clients in C

The gSOAP compiler can be used to create C (instead of C++) Web services and clients. The gSOAP stub and skeleton compiler soapcpp generates .cpp files by default. However, these files only use C syntax and data types if the header file input to soapcpp uses C syntax and data types. Therefore, a C compiler can be used to compile the .cpp files (e.g. by renaming the extensions to .c) to create C Web service and client executables. For example, with symbolic links on Unix/Linux:

```
lx -s soapC.cpp soapC.c
lx -s soapClient.cpp soapClient.c
```
5.5 Limitations of gSOAP

gSOAP is fully SOAP 1.1 (and partly SOAP 1.2) compliant and supports all SOAP 1.1 RPC features.

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 remote methods.

The following C++ language constructs cannot be used by the header file input to the SOAP C++ stub and skeleton compiler:

Templates The SOAP C++ stub and skeleton compiler is a preprocessor and cannot predict the template instantiations used by the main program, nor can it generate templated code.

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.

Pragmas All pragmas such as #include and #define are not supported. All pragmas are ignored by the compiler. A traditional C++ preprocessor can be used for the interpretation of pragmas. For example, Unix and Linux users can use “cpp -B” to expand the header file, e.g. cpp -B myfile.h | soapcpp.

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.

In addition, the following data types cannot be used in the header file (they can, however be used as a class method return type and as class method parameter types of a class declared in the header file):

union types Because the run-time value of a union data type cannot be determined by the compiler, the data type cannot be encoded. 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 encoded. 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.

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` [ ] cannot be used. gSOAP supports dynamic arrays using a special type convention, see Section 7.8.

**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 7.8.

5.6 **gSOAP Serialization Options and Flags**

Global flag variables can be set (i.e. 1, where default is 0 which means off) to control the SOAP XML serialization of data with gSOAP. These flags are global `int` variables and can be set anywhere in the client/service application, but before serialization takes place. Although gSOAP is fully SOAP 1.1 compliant, some SOAP implementations may have trouble accepting multi-reference data and implicit null data so these flags can be used to put gSOAP in “safe mode”. In addition, the embedding of multi-reference data is a feature that is likely to be adopted in future SOAP specifications which gSOAP already supports (turned off by default).

<table>
<thead>
<tr>
<th>Global Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>soap_disable_href</td>
<td>Do not serialize multi-reference data, but copy data in SOAP payload</td>
</tr>
<tr>
<td>soap_enable_embedding</td>
<td>Embed multi-reference data instead of encoding independent elements</td>
</tr>
<tr>
<td>soap_enable_null</td>
<td>Fault when xsi:nil=&quot;true&quot; attribute is received for a non-pointer data type (normally default value)</td>
</tr>
<tr>
<td>soap_enable_utf_string</td>
<td>Store and emit UTF8/16 encoded strings (<code>char*</code>) without translation</td>
</tr>
<tr>
<td>soap_disable_request_count</td>
<td>Do not include HTTP Content-Length in service request</td>
</tr>
<tr>
<td>soap_disable_response_count</td>
<td>Do not include HTTP Content-Length in service response (use this option for CGI applications as the Web server determines Content-Length)</td>
</tr>
<tr>
<td>soap_enable_array_overflow</td>
<td>Do not fault when receiving excess elements that do not fit in a fixed-size array</td>
</tr>
</tbody>
</table>

The flags can also be selectively turned on/off when multiple Web services are accessed by a client. The flags control the serialization only. Deserialization can handle different serialization formats automatically.

**Caution**: Disabling hrefs (multi-reference data output) can be used to improve interoperability with SOAP implementations that are not fully SOAP 1.1 compliant. However, disabling hrefs will crash the serializer for cyclic data structures.

5.7 **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 involves calls to the malloc library function 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.

gSOAP does not 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 deallocation functions.

The deallocation functions are:

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>soap_end()</td>
<td>Remove temporary data structures and deserialized data except class instances</td>
</tr>
<tr>
<td>soap_free()</td>
<td>Remove temporary data structures only</td>
</tr>
<tr>
<td>soap_destroy()</td>
<td>Remove all dynamically allocated class instances</td>
</tr>
<tr>
<td>soap_dealloc(NULL)</td>
<td>Remove all dynamically allocated deserialized data except class instances</td>
</tr>
<tr>
<td>soap_unlink(void *p)</td>
<td>Unlink data/object at p from gSOAP’s deallocation chain</td>
</tr>
</tbody>
</table>

Temporary data (i.e. the hash tables) are automatically removed with a call to the soap_free() function when the next call to a stub or skeleton routine is made.

There are two situations to consider for memory deallocation policies:

1. The client/service application does not use any class data structures that are (de)marshalled in SOAP, but uses structs, arrays, etc. In this case, calling the soap_end() function is safe to remove all deserialized data. The function can be called after processing the deserialized data of a remote method call or after a number of remote method calls have been made.

2. The client/service application uses class data structures. In this case, either:

   • 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,

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

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

There is one exception:

• A dynamic array class SHOULD delete the contents of the array it points to as part of its destructor’s operations (this includes classes for hexBinary and base64Binary schema types.

This exception was chosen because dynamic arrays (Section 7.8) form a class of special array constructs and deleting a class instance should amount to the deletion of the (deserialized) array. The array is not removed by the deallocation functions listed in the table.
Possible problematic cases arise when class data types are used mixed with other data structures such as structs. It is advised to use pointers to class instances that are used within structs or to use classes only (not structs). Structs that contain class instances may be malloced which does not initialize the class instances contained. Using a pointer in the struct to the class instance solves this problem because the class instance will be created and initialized on the heap. Also, when a copy of a class instance is made (e.g. for deserialization in a temporary structure such as a response element), the destructor of the instance copy will be invoked which can possibly do damage to the original instance through shared data parts. Therefore, it is advised to always pass class data types by pointer to a remote method. For example:

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

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

```c
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, as in:

```c
class X { ... };
class ArrayOfint { int *ptr; int _size; };
ns_remoteMethod(X *in, ArrayOfint *out);
```

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

```c
class X { ... };
class ArrayOfint { 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:

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void *soap_malloc(size_t n)</td>
<td>return pointer to n bytes</td>
</tr>
<tr>
<td>Class *soap_new_Class(int n)</td>
<td>instantiate n Class objects</td>
</tr>
</tbody>
</table>

The `soap_new_X` functions are generated by the gSOAP compiler for every class `X` in the header file. Parameter `n` MUST be -1 to instantiate a single object or ≥ 0 to instantiate an array of `n` objects.

Space allocated with `soap_malloc` will be released with the `soap_end` and `soap_dealloc` functions. The objects instantiated with `soap_new_X` are removed with `soap_destroy`. For example, the following service uses temporary data in the remote method implementation:

```c
int main()
{
    ...
    soapServe();
    soapEnd();
    ...
}
```
An example remote method that allocates a temporary string is:

```c
int ns__itoa(int i, char **a)
{
  *a = (char*)soap_malloc(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:

```c
int ns__mymethod(...)
{
  ...
  if (exception)
  {
    soap_fault.faultstring = (char*)soap_malloc(1024);
    strcpy(soap_fault.faultstring, ...);
    return SOAP_FAULT;
  }
  ...
}
```

### 5.8 Debugging

To activate message logging for debugging, un-comment `#define DEBUG` pragma in `stdsoap.h`. Compile the client and/or server applications as described above (or simply use `g++ -DDEBUG ...` to compile with debugging activated). When the client and server applications run, they will log their activity in three separate files:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SENT.log</td>
<td>The SOAP content transmitted by the application</td>
</tr>
<tr>
<td>RECV.log</td>
<td>The SOAP content received by the application</td>
</tr>
<tr>
<td>TEST.log</td>
<td>A log containing various activities performed by the application</td>
</tr>
</tbody>
</table>

**Caution:** The client and server applications may run very slowly due to the logging activity.

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,

```bash
myservice.cgi < SENT.tst
```

This should display the service response on the terminal.

**Caution:** Turn debugging off when installing the CGI application on the Web. Most Web servers do not allow the creation of the log files and the CGI application will be terminated resulting in an HTTP error send to the client.
5.9 Libraries

- The socket library is essential and requires the inclusion of the appropriate libraries with the compile command for Sun Solaris systems:

  g++ -o myclient myclient.cpp stdsoap.cpp soapC.cpp soapClient.cpp -lsocket -lxnet -lnsl -lm

  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 stdsoap.h header file. Then, the application can be linked without the -lm math library under Sun Solaris:

  g++ -o myclient myclient.cpp stdsoap.cpp soapC.cpp soapClient.cpp -lsocket -lxnet -lnsl

6 The gSOAP Remote Method Specification Format

A SOAP remote method 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 6.2. Multiple remote methods MAY be declared together in one header file.

The general form of a SOAP remote method 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 6.3)
method_name is the remote method name (see identifier translation rules 6.3)
inparam is the declaration of an input parameter of the remote method
outparam is the declaration of the output parameter of the remote method

This simple form can only pass a single, non-struct and non-class type output parameter. See 6.1 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 6.3. (The optional parts are shown enclosed in \[\].)

The XML response by the Web service is of the form:

```xml
<\[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 6.3. By convention, the response element name is the method name ending in Response. See 6.1 on how to change the declaration if the service response element name is different.

The gSOAP stub and skeleton compiler generates a stub routine and a proxy for the remote method. This proxy is of the form:

```c
int soap_call_{namespace_prefix__}method_name(char *URL, char *action, [inparam1, inparam2, ...] outparam);
```

This proxy can be called by a client application to perform the remote method call.

The gSOAP stub and skeleton compiler generates a skeleton routine for the remote method. The skeleton function is:

```c
int soap_serve_{namespace_prefix__}method_name();
```

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.

### 6.1 Remote Method Parameter Passing

The input parameters of a remote method MUST be passed by value. Input parameters cannot be passed by reference with the & reference operator, but an input parameter value MAY be passed using a pointer to the data. 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 remote method 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 5.5

If the output parameter is a struct or class type, it is considered a SOAP remote method 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
remote method, see also 4.1.6. 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 remote method returns multiple output parameters, a response struct or class MUST be declared. By convention, the response element is the remote method 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 6.3)
response_element_name it the name of the response element (see identifier translation rules 6.3)
outparam is the declaration of an output parameter of the remote method

The general form of a remote method 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 6.3. (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 6.3. (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 4.1.12.

### 6.2 Stub and Skeleton Routine Error Codes

The error codes returned by the stub and skeleton routines are listed below.

<table>
<thead>
<tr>
<th>#</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SOAP_OK</td>
<td>No error</td>
</tr>
<tr>
<td>1</td>
<td>SOAP_CLI_FAULT*</td>
<td>The service raised a client fault exception</td>
</tr>
<tr>
<td>2</td>
<td>SOAP_SVR_FAULT*</td>
<td>The service raised a server fault exception</td>
</tr>
<tr>
<td>3</td>
<td>SOAP_TAG_MISMATCH</td>
<td>An XML element didn’t correspond to anything expected</td>
</tr>
<tr>
<td>4</td>
<td>SOAP_TYPE_MISMATCH</td>
<td>An XML schema type mismatch</td>
</tr>
<tr>
<td>5</td>
<td>SOAP_SYNTAX_ERROR</td>
<td>An XML syntax error occurred on the input</td>
</tr>
<tr>
<td>6</td>
<td>SOAP_NO_TAG</td>
<td>Begin of an element expected, but not found</td>
</tr>
<tr>
<td>7</td>
<td>SOAP_JOB</td>
<td>Array index out of bounds</td>
</tr>
<tr>
<td>8</td>
<td>SOAP_MUSTUNDERSTAND*</td>
<td>An element needs to be ignored that need to be understood</td>
</tr>
<tr>
<td>9</td>
<td>SOAP_NAMESPACE</td>
<td>Namespace name mismatch (validation error)</td>
</tr>
<tr>
<td>10</td>
<td>SOAP_OBJ_MISMATCH</td>
<td>Mismatch in the size and/or shape of an object</td>
</tr>
<tr>
<td>11</td>
<td>SOAP_FATAL_ERROR</td>
<td>Internal error</td>
</tr>
<tr>
<td>12</td>
<td>SOAP_FAULT</td>
<td>An exception was raised by the service</td>
</tr>
<tr>
<td>13</td>
<td>SOAP_NO_METHOD</td>
<td>Skeleton error: the skeleton cannot serve the method</td>
</tr>
<tr>
<td>14</td>
<td>SOAP_EOM</td>
<td>Out of memory</td>
</tr>
<tr>
<td>15</td>
<td>SOAP_NULL</td>
<td>An element was null, while it is not supposed to be null</td>
</tr>
<tr>
<td>16</td>
<td>SOAP_MULTI_ID</td>
<td>Multiple occurrences of the same element ID on the input</td>
</tr>
<tr>
<td>17</td>
<td>SOAP_MISSING_ID</td>
<td>Element ID missing for an HREF on the input</td>
</tr>
<tr>
<td>18</td>
<td>SOAP_HREF</td>
<td>Reference to object is incompatible with the object refered to</td>
</tr>
<tr>
<td>19</td>
<td>SOAP_TCP_ERROR</td>
<td>A TCP connection error occured</td>
</tr>
<tr>
<td>20</td>
<td>SOAP_HTTP_ERROR</td>
<td>An HTTP error</td>
</tr>
<tr>
<td>21</td>
<td>SOAP_SSL_ERROR</td>
<td>An SSL error</td>
</tr>
<tr>
<td>22</td>
<td>SOAP_VERSIONMISMATCH*</td>
<td>SOAP version mismatch or no SOAP message</td>
</tr>
<tr>
<td>-1</td>
<td>SOAP_EOF</td>
<td>Unexpected end of file</td>
</tr>
</tbody>
</table>

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 remote method call was successful (the SOAP_OK predefined constant is guaranteed to be 0). The error code is also stored in the global `soap_error` variable. The function `soap_print_fault(stderr)` can be called to display an error message on `stderr` where current value of the `soap_error` variable is used by the function to display the error. The function `soap_print_fault_location(stderr)` prints the location of the error if the error is a result from parsing XML.

A remote method 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 to the strings `soap_fault.faultstring` and `soap_fault.detail`, see Section 8.
6.3 C++ Identifier Name to XML Element Name Translation

One of the secrets behind the power and flexibility of gSOAP’s encoding and decoding of remote method 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
```

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 6.4, see also Section 4.1.2. The namespace URI is a unique identification that can be associated with the remote methods and data types. The namespace URI disambiguates potentially identical remote method names and data type names used by disparate organizations.

XML element names are NCNames (restricted strings) that MAY contain hypens, dots, and underscores. The special characters in the XML element names of remote methods, 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_DOT_com
{
    char *n_s_:biz_USCORE_name;
};
```

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 structs and classes with essentially the same XML schema type name, but that have to be distinguished as separate 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._DOT_.com matches the element name I-want:SOAP4fun@the-beach.com.
6.4 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:

```c
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/1999/XMLSchema-instance", // MUST be third
   "xsd", "http://www.w3.org/1999/XMLSchema"}, // Required for XML schema types
  {"ns1", "urn:my-service-URI"}, // The namespace URI of the remote methods
  {NULL, NULL} // end of table
};
```

Each namespace prefix used by an identifier name in the header file specification (see Section 6.3) 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 wildcards. 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:

```c
struct Namespace namespaces[] =
{
  // ("ns-prefix", "ns-name", "ns-name validation pattern")
  ...
  {"xsi", "http://www.w3.org/1999/XMLSchema-instance", "http://www.w3.org/-----/XMLSchema-instance"},
  {"xsd", "http://www.w3.org/1999/XMLSchema", "http://www.w3.org/-----/XMLSchema"},
  ...
```

Or alternatively, asterisks can be used as wildcards for multiple characters:

```c
struct Namespace namespaces[] =
{
  // ("ns-prefix", "ns-name", "ns-name validation pattern")
  ...
  {"xsd", "http://www.w3.org/1999/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 remote method 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 %\{URI\}% in the table. See Section 10.1 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.

For example, let's say we have the following structs:

```c
struct a_elt { ... }
struct b_elt { ... }
struct k_elt { ... }
```

and a namespace mapping table in the program:

```c
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:

```xml
<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
... 
```

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 remote method 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 remote method invocation. For example:

```c
struct Namespace namespacesTable1[] = { ... }
struct Namespace namespacesTable2[] = { ... }
struct Namespace namespacesTable3[] = { ... }
```
struct Namespace *namespaces;
...
namespaces = namespaceTable1;
soap_call_remote_method(URL, Action, ...);
...

7 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.

7.1 Primitive Type Encoding

The default encoding rules for the primitive C and C++ data types are given in the table below:

<table>
<thead>
<tr>
<th>Type</th>
<th>Schema Type</th>
<th>Example Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>boolean</td>
<td>&lt;boolean xsi:type=&quot;boolean&quot;&gt;...&lt;/boolean&gt;</td>
</tr>
<tr>
<td>char* (C string)</td>
<td>string</td>
<td>&lt;string xsi:type=&quot;string&quot;&gt;...&lt;/string&gt;</td>
</tr>
<tr>
<td>char</td>
<td>byte</td>
<td>&lt;byte xsi:type=&quot;byte&quot;&gt;...&lt;/byte&gt;</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>&lt;double xsi:type=&quot;double&quot;&gt;...&lt;/double&gt;</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>&lt;float xsi:type=&quot;float&quot;&gt;...&lt;/float&gt;</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>&lt;int xsi:type=&quot;int&quot;&gt;...&lt;/int&gt;</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>&lt;long xsi:type=&quot;long&quot;&gt;...&lt;/long&gt;</td>
</tr>
<tr>
<td>LONG64</td>
<td>long</td>
<td>&lt;long xsi:type=&quot;long&quot;&gt;...&lt;/long&gt;</td>
</tr>
<tr>
<td>long long</td>
<td>long</td>
<td>&lt;long xsi:type=&quot;long&quot;&gt;...&lt;/long&gt;</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
<td>&lt;short xsi:type=&quot;short&quot;&gt;...&lt;/short&gt;</td>
</tr>
<tr>
<td>time_t</td>
<td>dateTime</td>
<td>&lt;dateTime xsi:type=&quot;dateTime&quot;&gt;...&lt;/dateTime&gt;</td>
</tr>
<tr>
<td>unsigned char</td>
<td>unsignedByte</td>
<td>&lt;unsignedByte xsi:type=&quot;unsignedByte&quot;&gt;...&lt;/unsignedByte&gt;</td>
</tr>
<tr>
<td>unsigned int</td>
<td>unsignedInt</td>
<td>&lt;unsignedInt xsi:type=&quot;unsignedInt&quot;&gt;...&lt;/unsignedInt&gt;</td>
</tr>
<tr>
<td>unsigned long</td>
<td>unsignedLong</td>
<td>&lt;unsignedLong xsi:type=&quot;unsignedLong&quot;&gt;...&lt;/unsignedLong&gt;</td>
</tr>
<tr>
<td>ULONG64</td>
<td>unsignedLong</td>
<td>&lt;unsignedLong xsi:type=&quot;unsignedLong&quot;&gt;...&lt;/unsignedLong&gt;</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>unsignedLong</td>
<td>&lt;unsignedLong xsi:type=&quot;unsignedLong&quot;&gt;...&lt;/unsignedLong&gt;</td>
</tr>
<tr>
<td>wchar*</td>
<td>string</td>
<td>&lt;string xsi:type=&quot;string&quot;&gt;...&lt;/string&gt;</td>
</tr>
</tbody>
</table>

Objects of type void and void* cannot be encoded.

7.2 How to Encode and Decode Primitive Types as Built-In XML Schema 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 with an optional namespace prefix by using a typedef in the header file input to the gSOAP stub and skeleton compiler. The declaration enables the implementation of built-in XML schema 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 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 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 XML schema types such as `xsd:anyURI`, `positiveInteger`, and `decimal` can be stored by standard primitive data structures in C++ as well such as strings, integers, floats, and doubles. To serialize strings, integers, floats, and doubles as built-in primitive and derived XML schema types. To this end, a `typedef` declaration can be used to declare an XML Schema type.

For example, the declaration

```c
typedef unsigned int positiveInteger;
```

creates a named type `positiveInteger` which is represented by `unsigned int` in C++. For example, the encoding of a `positiveInteger` value 3 is

```xml
<positiveInteger xsi:type="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:

  ```c
typedef char *xsd__anyURI;
```

- **xsd:base64Binary** Represents Base64-encoded arbitrary binary data. For using the `xsd:base64Binary` XML schema type, the use of the base64Binary representation of a dynamic array is strongly recommended, see Section 7.9. However, the type can also be declared as a string and the encoding will be string-based:

  ```c
typedef char *xsd__base64Binary;
```

  With this approach, it is solely 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` XML 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 7.3.5. The corresponding type declaration is:

  ```c
typedef bool xsd__boolean;
```

  Type `xsd__boolean` declares a Boolean (0 or 1), which is encoded as

  ```xml
  <xsd:boolean xsi:type="xsd:boolean">...</xsd:boolean>
  ```
xsd:byte Represents a byte (-128...127). The corresponding type declaration is:

```c
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 XML schema types and the type declaration is:

```c
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 below this range, the time_t value will be set to -2147483648, and upon receiving a date above this range, the time_t value will be set to 2147483647. Strings (char*) can be used to store xsd:dateTime XML schema types. The type declaration is:

```c
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 XML schema types. The type declaration is:

```c
typedef char *xsd_date;
```

xsd:decimal Represents arbitrary precision decimal numbers. It is recommended to use the double type to store xsd:decimal XML schema types and the type declaration is:

```c
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:

```c
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 XML schema types. The type declaration is:

```c
typedef char *xsd_duration;
```

xsd:float Corresponds to the IEEE single-precision 32-bit floating point type. The type declaration is:

```c
typedef float xsd_float;
```

Type xsd_float declares a floating point number which is encoded as

```xml
<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 11110110111. For using the xsd:hexBinary XML schema type, the use of the hexBinary representation of a dynamic array is strongly recommended, see Section 7.10. However, the type can also be declared as a string and the encoding will be string-based:

```c
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:

```c
typedef int xsd_int;
```

Otherwise, the C++ compiler supports 16-bit int types and the type declaration should use the long type:

```c
typedef long xsd_int;
```

Type xsd_int declares a 32-bit integer which is encoded as

```xml
<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:

```c
typedef long long xsd_integer;
```

Type xsd_integer declares a 64-bit integer which is encoded as an unbounded xsd:integer:

```xml
<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:normalizedString` XML 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 -323768 to 323767. 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 XML 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 XML schema types. The type declaration is:

```c
typedef char *xsd_token;
```

Type xsd_token declares a string type which is encoded as

```xml
<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:

```c
typedef unsigned char xsd_unsignedByte;
```

Type xsd_unsignedByte declares an unsigned 8-bit integer which is encoded as

```xml
<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:

```c
typedef unsigned int xsd unsignedInt;
```

Otherwise, the C++ compiler supports 16-bit int types and the type declaration should use the long type:

```c
typedef unsigned long xsd_unsignedInt;
```

Type xsd_unsignedInt declares an unsigned 32-bit integer which is encoded as

```xml
<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:

```c
typedef unsigned long long xsd_unsignedLong;
```

Or in Visual C++:

```c
typedef ULONG64 xsd_unsignedLong;
```

Type xsd_unsignedLong declares an unsigned 64-bit integer which is encoded as

```xml
<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:

```c
typedef unsigned short xsd unsignedShort;
```

Type xsd_unsignedShort declares an unsigned short 16-bit integer which is encoded as
Other XML schema types such as gYearMonth, gYear, gMonthDay, gDay, xsd:gMonth, QName, NOTATION, etc., can be encoded similarly using a typedef declaration.

7.2.1 How to Specify Multiple Storage Formats for a Single Primitive XML Schema Type

Trailing underscores (see Section 6.3) 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:

```c
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.

7.2.2 How to Specify Polymorphic Primitive Types

SOAP 1.1 supports polymorphic types, because XML schema types form a hierarchy. The root of the hierarchy is called xsd:anyType. 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++ classes 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:

```c
class xsd_type_name: [public xsd_super_type_name]
{ public: Type_item;
  [public] [private] [protected] method1;
  method2;
  ...
};
```

where Type is a primitive C type that may be declared with a typedef to enforce XML schema encoding as with the usual typedef conventions used by the gSOAP compiler.

For example, the XML schema type hierarchy can be copied to C++ with the following declarations:
typedef char *xsd;
class xsd
*
class xsd

typedef char *xsd;
class xsd
*
* xsd;
class xsd

typedef double xsd;
class xsd
*
* xsd;
class xsd

typedef double xsd;
class xsd
*
* xsd;
class xsd

typedef float xsd;
class xsd
*
* xsd;
class xsd

typedef float xsd;
class xsd
*
* xsd;
class xsd

typedef float xsd;
class xsd
*
* xsd;
class xsd

typedef unsigned short xsd;
class xsd
*
* xsd;
class xsd

typedef unsigned short xsd;
class xsd
*
* xsd;
class xsd

typedef unsigned char xsd;
class xsd
*
* xsd;
class xsd

typedef unsigned char xsd;
class xsd
*
* xsd;
class xsd

typedef unsigned short xsd;
class xsd
*
* xsd;
class xsd

typedef unsigned short xsd;
class xsd
*
* xsd;
class xsd

typedef unsigned short xsd;
class xsd
*
* xsd;
class xsd

typedef unsigned char xsd;
class xsd
*
* xsd;
class xsd

**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 7.9 and 7.10.

Methods are allowed to be added to the classes above, such as constructors and getter/setter methods.

### 7.2.3 XML Schema Type Decoding Rules

The decoding rules for the primitive C and C++ data types is given in the table below:
<table>
<thead>
<tr>
<th>Type</th>
<th>Allows Decoding of</th>
<th>Precision Lost?</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>xsd:boolean</td>
<td>yes</td>
</tr>
<tr>
<td>char* (C string)</td>
<td>any type, see 7.2.5</td>
<td>yes</td>
</tr>
<tr>
<td>wchar_t* (wide string)</td>
<td>any type, see 7.2.5</td>
<td>yes</td>
</tr>
<tr>
<td>double</td>
<td>xsd:double</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:float</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:long</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:int</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:short</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:byte</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:unsignedLong</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:unsignedInt</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:unsignedShort</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:unsignedByte</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:decimal</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd:integer</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd:positiveInteger</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd:negativeInteger</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd:nonPositiveInteger</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd:nonNegativeInteger</td>
<td>possibly</td>
</tr>
<tr>
<td>float</td>
<td>xsd:float</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:long</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:int</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:short</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:byte</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:unsignedLong</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:unsignedInt</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:unsignedShort</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:unsignedByte</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:decimal</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd:integer</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd:positiveInteger</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd:negativeInteger</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd:nonPositiveInteger</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd:nonNegativeInteger</td>
<td>possibly</td>
</tr>
<tr>
<td>long long</td>
<td>xsd:long</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:int</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:short</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:byte</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>xsd:unsignedLong</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd:unsignedInt</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd:unsignedShort</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd:unsignedByte</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd:integer</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd:positiveInteger</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd:negativeInteger</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd:nonPositiveInteger</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd:nonNegativeInteger</td>
<td>possibly</td>
</tr>
<tr>
<td>Type</td>
<td>Allows Decoding of</td>
<td>Precision Lost?</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>long</td>
<td>xsd: long</td>
<td>possibly, if long is 32 bit</td>
</tr>
<tr>
<td></td>
<td>xsd: int</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: short</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: byte</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedLong</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedInt</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedShort</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedByte</td>
<td>no</td>
</tr>
<tr>
<td>int</td>
<td>xsd: int</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: short</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: byte</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedInt</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedShort</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedByte</td>
<td>no</td>
</tr>
<tr>
<td>short</td>
<td>xsd: short</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: byte</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedShort</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedByte</td>
<td>no</td>
</tr>
<tr>
<td>char</td>
<td>xsd: byte</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedByte</td>
<td>possibly</td>
</tr>
<tr>
<td>unsigned long</td>
<td>xsd: unsignedLong</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedInt</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedShort</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedByte</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: positiveInteger</td>
<td>possibly</td>
</tr>
<tr>
<td></td>
<td>xsd: nonNegativeInteger</td>
<td>possibly</td>
</tr>
<tr>
<td>unsigned long</td>
<td>xsd: unsignedLong</td>
<td>possibly, if long is 32 bit</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedInt</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedShort</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedByte</td>
<td>no</td>
</tr>
<tr>
<td>unsigned int</td>
<td>xsd: unsignedInt</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedShort</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedByte</td>
<td>no</td>
</tr>
<tr>
<td>unsigned short</td>
<td>xsd: unsignedShort</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xsd: unsignedByte</td>
<td>no</td>
</tr>
<tr>
<td>unsigned char</td>
<td>xsd: unsignedByte</td>
<td>no</td>
</tr>
<tr>
<td>time_t</td>
<td>xsd: dateTime</td>
<td>no(?)</td>
</tr>
</tbody>
</table>

Due to limitations in representation of certain primitive C++ types, a possible loss of accuracy may occur with the decoding of certain XML 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 XML schema encoded primitive types adhere to the same decoding rules. For example, the following declaration:

```c
typedef unsigned long long xsd_nonNegativeInteger;
```

enables the encoding and decoding of `xsd:nonNegativeInteger` XML schema types (although decoding takes place with a possible loss of precision). The declaration also allows decoding of `xsd:positiveInteger` XML schema types, because of the storage as an `unsigned long long` data type.

### 7.2.4 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

```c
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`:

```xml
<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

```c
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.

### 7.2.5 “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` remote method returns some detailed information. The remote method is declared as:

```c
// Contents of header file "getInfo.h"
getInfo(char *detail):
```

67
The proxy of the remote method 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

xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/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>”.

### 7.2.6 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 the static `soap_double_format` string variable. For example:

```
soap_double_format = "%e";
```

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:

```
soap_float_format = "%.4f";
```

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.

### 7.2.7 INF, -INF, and NaN Values of float and double Types

The gSOAP runtime `stdsoap.cpp` and header file `stdsoap.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 `stdsoap.h` file. You can control the representations as well, which are defined by the macros:
7.3 Enumeration Type Encoding and Decoding

Enumerations are generally useful for the declaration of named integer-valued constants, also called enumeration constants.

7.3.1 Symbolic Encoding of Enumeration Constants

The gSOAP stub and skeleton compiler encodes the constants of enumeration-typed variables in symbolic form using the names of the constants when possible to comply to SOAP’s XML schema enumeration encoding style. Consider for example the following enumeration of weekdays:

```c
enum weekday {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

The enumeration-constant `Mon`, for example, is encoded as

```xml
<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:

```c
enum ns1_weekday {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

The enumeration-constant `Sat`, for example, is encoded as:

```xml
<ns1:weekday xsi:type="ns1:weekday">Sat</ns1:weekday>
```

The corresponding XML schema for this enumeration data type would be:

```xml
<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>
```
7.3.2 Literal 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:

```c
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:

```xml
<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 stub and skeleton compiler).

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:

```c
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.

7.3.3 Initialized Enumeration Constants

The gSOAP compiler supports the initialization of enumeration constants, as in:

```c
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.

7.3.4 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:
\begin{verbatim}
enum ns1._workday {Mon, Tue, Wed, Thu, Fri};
enum ns1._weekday {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
\end{verbatim}

which will result in the encoding of the constants of \texttt{enum ns1._weekday} without the underscore, for example as \texttt{Mon}.

\textbf{Caution}: The following declaration:
\begin{verbatim}
enum ns1._workday {Mon, Tue, Wed, Thu, Fri};
enum ns1._weekday {Sat = 5, Sun = 6};
\end{verbatim}

will not properly encode the \texttt{weekday} enumeration, because it lacks the named constants for \texttt{workday} in its enumeration list.

\section*{7.3.5 Boolean Enumeration Type Encoding and Decoding for C Compilers}

When a pure C compiler is used to create SOAP clients and services, the \texttt{bool} type may not be supported by the compiler and in that case an enumeration type should be used. The C enumeration-type encoding adopted by the gSOAP stub and skeleton compiler can be used to encode boolean values according to the SOAP encoding style. The namespace prefix can be specified with the usual namespace prefix convention for identifiers to explicitly specify the encoding style.

For example, the built-in \texttt{boolean} XML schema type supports the mathematical concept of binary-valued logic. The \texttt{boolean} XML schema encoding style can be specified by using the \texttt{xsd} prefix. For example:
\begin{verbatim}
enum xsd._boolean {false_, true_};
\end{verbatim}

The value \texttt{false_}, for example, is encoded as:
\begin{verbatim}
<xsd:boolean xsi:type="xsd:boolean">false</xsd:boolean>
\end{verbatim}

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 \texttt{false} and \texttt{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:
\begin{verbatim}
enum SOAP_ENC._boolean {};
\end{verbatim}

The value 0, for example, is encoded with an integer literal:
\begin{verbatim}
<SOAP-ENC:boolean xsi:type="SOAP-ENC:boolean">0</SOAP-ENC:boolean>
\end{verbatim}

\section*{7.3.6 Bitmask Enumeration Encoding and Decoding}

A bitmask is an enumeration of flags such as declared with C\#’s \texttt{[Flags]} \texttt{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 compiler to indicate the use of bitmasks with an asterisk:
The gSOAP 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:

```c
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 compiler generates the enumeration:

```c
enum ns._machineStatus { ON=1, BELT=2, VALVE=4, HATCH=8};
```

A remote method implementation in a Web service can return:

```c
int ns._getMachineStatus(char *name, enum ns._machineStatus result)
{
    *result = BELT — HATCH;
    return SOAP_OK;
}
```

### 7.4 struct Encoding and Decoding

A `struct` data type is encoded as a SOAP 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 7.5 for further details. However, the encoding and decoding of `structs` is more efficient compared to `class` instances due to the lack of inheritance and the requirement by the marshalling routines to check inheritance properties at run time.

### 7.5 class Instance Encoding and Decoding

A `class` instance is encoded as a SOAP 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:

```c
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 6.3)

class_name1 is the element name of the compound data type (see identifier translation rules 6.3).

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 and have no effect on the declaration and MAY therefore be omitted. All access permissions are converted to public by the gSOAP stub and skeleton compiler.

A class name is REQUIRED to be unique and cannot have the same name as a struct, enum, or remote method name specified in the header file input to the gSOAP compiler. The reason is that remote method 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 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.

Templates are not supported by the gSOAP compiler version 1.2.3.

A class instance is encoded as:

```xml
<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.
7.5.1 Example

The following example declares a base class `ns::Object` and a derived class `ns::Shape`:

```cpp
// 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:

```xml
<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 6.4.

7.5.2 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:

```cpp
// 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:

```xml
<ns:Triangle xsi:type="ns:Triangle">
  <name xsi:type="string">Triangle</name>
</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 in the future.

### 7.5.3 Class Methods

A **class** declaration in the header file input to the gSOAP 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**.

### 7.5.4 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 remote method parameters. A client application MAY use pointers to instances of derived classes for the input parameters of a remote method. If the service was compiled with a declaration and implementation of the derived class, the remote method 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 class and when a service returns an instance of a derived class.

The following example declares Base and Derived classes and a remote method that takes a pointer to a Base class instance and returns a Base class instance:

```cpp
// 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 compiler to produce the stub and skeleton routines which are used to implement a client and service. The pointer of the remote method 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:

```cpp
// 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 remote method:

```cpp
// CLIENT
#include "soapH.h"
main()
{
    Derived obj1;
    Base *obj2;
    struct methodResponse r;
    obj1.name = "X";
    obj1.num = 3;
    soap_call_method(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:

```cpp
// SERVER1
#include "soapH.h"
main()
{
    soap_server();
}
int method(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:

```cpp
// 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 stub and skeleton compiler 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:

```cpp
// 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:

```cpp
// SERVER2
#include "soapH.h"
main()
{
  soap_server();
}
```c

int method(Base *obj1, struct methodResponse &result) {
    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.

### 7.6 Pointer Encoding and Decoding

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.

#### 7.6.1 Multi-Reference 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` attributes to refer to the multi-reference data (also see Section 5.6 on options to control the serialization of multi-reference data). Cyclic C/C++ data structures are encoded with multi-reference SOAP encoding. Consider for example the following a linked list data structure:

```c
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:

```xml
<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>
</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 structs are similar, except that the first uses pointer-based fields while the other uses non-pointer-based fields:

```c
typedef long xsd_int;
struct ns_record
{
    xsd_int *a;
    xsd_int *b;
} P;
struct ns_record
{
    xsd_int a;
    xsd_int b;
} R;
```

Since both `a` and `b` fields of `P` point to the same integer, the encoding of `P` is multi-reference:

```xml
<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 structs resemble the same XML data type because the trailing underscore will be ignored in XML encoding and decoding.

### 7.6.2 NULL Pointers and Nil Elements

A NULL pointer is not serialized, unless the pointer itself is pointed to by another pointer (but see Section 5.6 to control the serialization of NULLs). For example:

```c
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

```xml
<... 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_enable_null flag is set. The types section of a WSDL description contains information on the “nilability” of data.

### 7.7 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]">
        <float xsi:type="float">...</float>
        <float xsi:type="float">...</float>
        <float xsi:type="float">...</float>
    </SOAP-ENC:Array>
        <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.

### 7.8 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
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.

### 7.8.1 One-Dimensional Dynamic Arrays

A special form of struct or class is used for one-dimensional dynamic arrays that contains 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 for one-dimensional dynamic arrays is:

```c
struct some_name
{
    Type *__ptr;
    int __size;
    [[static const] int __offset [= ...];]
    ... // 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 typedefed 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 list/vector (de)serialization is used (see Section 7.8.6).

An alternative is to use a class with optional methods:

```c
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 SOAP list/vector, see Section 7.8.6.

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.

### 7.8.2 Example

The following example header file specifies the XMethods Service Listing service `getAllSOAPServices` remote method and an array of SOAPService data structures:
class ns3::SOAPService
{
    public:
    int ID;
    char *name;
    char *owner;
    char *description;
    char *homepageURL;
    char *endpoint;
    char *SOAPAction;
    char *methodNamespaceURI;
    char *serviceStatus;
    char *methodNamespace;
    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 ns3::getAllSOAPServices(ServiceArray &return_);

An example client application:

#include "soapH.h" ...

// ServiceArray class method implementations:
ServiceArray::ServiceArray()
{
    _ptr = NULL;
    _size = 0;
}

ServiceArray::~ServiceArray()
{
    if (_ptr)
        free(_ptr);
    _size = 0;
}

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:
{
    ServiceArray result;
    const char *endpoint = "www.xmethods.net:80/soap/servlet/rpcrouter";
    const char *action = "urn:xmethodsServicesManager#getAllSOAPServices";
    ...
    soap_call_ns__getAllSOAPServices(endpoint, action, result);
    result.print();
    ... 
}

7.8.3 One-Dimensional Dynamic Arrays With Non-Zero Offset

The declaration of a dynamic array as described in 7.8 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.

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() doubles
{
    if (_._ptr)
        free(_._ptr);
}
float& Vector::operator[](int i)
An example program fragment that serializes a vector of 3 elements:

```c
Vector v(3);
    v[1] = 1.0;
    v[2] = 2.0;
    v[3] = 3.0;
soap_begin();
    v.serialize();
    v.put("vec");
soap_end();
```

The output is a partially transmitted array:

```
    <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.

### 7.8.4 Nested One-Dimensional Dynamic Arrays

One-dimensional dynamic arrays MAY be nested. For example, using class `Vector` declared in the previous section, class `Matrix` is declared:

```c
// 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.
7.8.5 Multi-Dimensional Dynamic Arrays

The general form of the `struct` declaration for K-dimensional \((K \geq 1)\) dynamic arrays is:

```plaintext
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 list/vector (de)serialization is used (see Section 7.8.6).

An alternative is to use a `class` with optional methods:

```plaintext
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 SOAP list/vector, see Section 7.8.6.

The deserializer of a dynamic array can decode partially transmitted multi-dimensional arrays.

For example, the following declaration specifies a matrix class:

```plaintext
typedef double xsd_double;
class Matrix
{
    public:
    xsd_double *-_ptr;
    int _size[2];
    int _offset[2];
};
```

In contrast to the matrix class of Section 7.8.4 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.
7.8.6 Dynamic Array as List Encoding

In case the name of the `struct` or `class` of a dynamic array has a namespace prefix, the data type is considered a list (a.k.a. vector) and will be serialized as a SOAP list and not encoded as a SOAP array.

For example:

```c
struct ns_Map
{
    struct ns_Binding {char *key; char *val; } *__ptr;
    int __size;
};
```

This declares a dynamic array, but the array will be serialized and deserialized as a list. For example:

```xml
<ns:Map xsi:type="ns:Map">
    <ns:Binding xsi:type="ns:Binding">
        <key>Joe</key>
        <val>555 77 1234</val>
    </ns:Binding>
    <ns:Binding xsi:type="ns:Binding">
        <key>Susan</key>
        <val>555 12 6725</val>
    </ns:Binding>
    <ns:Binding xsi:type="ns:Binding">
        <key>Pete</key>
        <val>555 99 4321</val>
    </ns:Binding>
</ns:Map>
```

Deserialization is less efficient compared to an array, because the size of the list 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.

A list (de)serialization is also in affect for dynamic arrays when the pointer field does not refer to a type that is associated with a schema. For example:

```c
struct vector
{
    int *__ptr;
    int __size;
};
```

Since `int` has no association with a schema, a `vector` structure `X` is serialized as:

```xml
<X>
    <item>1</item>
    <item>-2</item>
    ...
</X>
```
7.8.7 Polymorphic Dynamic Arrays and Lists

An array of pointers to class instances allows the encoding of polymorphic arrays (arrays of polymorphic element types) and lists. For example:

```cpp
class ns__Object {
    public:
    ...
};
class ns__Data: public ns__Object {
    public:
    ...
};
class ArrayOfObject {
    public:
    ns__Object **__ptr;
    int __size;
    int __offset;
};
```

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.

7.8.8 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:

```cpp
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 6.3. The default XML element tag name for array elements is `item` (which corresponds to the use of field name `__ptritem`).

Consider for example:

```cpp
struct ArrayOfstring {
    xsd_string *__ptrstring; int __size;
};
```

The array is serialized as:

```xml
<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.

### 7.8.9 Embedded Arrays and Lists

An array (or list) can be embedded in a struct/class without the need to declare a separate array data type. When a struct or class type declaration contains a `int _size` field and the next field below is a pointer type, gSOAP assumes the pointer type points to an array of values where the `_size` field holds the number of values at run time. Multiple arrays can be embedded in a struct/class by using `_size` field names that end with a unique name suffix.

The general convention for embedding arrays is:

```c
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.

For example, the following struct has two embedded arrays:

```c
struct ns._Contact
{
    char *firstName;
    char *lastName;
    int _sizePhones;
    ULONG64 *phoneNumber; // array of phone numbers
    int _sizeEmails;
    char **emailAddress; // array of email addresses
    char *socSecNumber;
};
```

The XML serialization of an example `ns._Contact` is:

```xml
<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>9999999999</socSecNumber>
</mycontact>
```
7.9 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:

```c
struct xsd_base64Binary
{
  unsigned char *_ptr;
  int _size;
};
```

Or with a class:

```c
class xsd_base64Binary
{
  public:
  unsigned char *_ptr;
  int _size;
};
```

When compiled by the gSOAP stub and skeleton compiler, 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:

```c
struct SOAP_ENC_base64
{
  unsigned char *_ptr;
  int _size;
};
```

Or with a class:

```c
class SOAP_ENC_base64
{
  unsigned char *_ptr;
  int _size;
};
```

When compiled by the gSOAP stub and skeleton compiler, 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:

```c
class xsd_base64Binary
{
  public:
  unsigned char *_ptr;
```
int _size;
xsd__base64Binary(); // Constructor
xsd__base64Binary(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(int n)
{
    _ptr = (unsigned int*)malloc(n);
    _size = n;
}
xsd__base64Binary::~xsd__base64Binary()
{
    if (_ptr)
        free(_ptr);
}
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", "r");  
xsd__base64Binary image(filesize(fd));  
fread(image.location(), image.size(), 1, fd);  
fclose(fd);  
soap_begin();  
image.soap_serialize();  
image.soap_put("jpegimage", NULL);  
soap_end();  
...

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;
The **struct** or **class** name `soap_enc_base64` should be used for `SOAP-ENC:base64` schema type instead of `xsd_base64Binary`.

### 7.10 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**:

```c
struct xsd_hexBinary {
    unsigned char * _ptr;
    int _size;
};
```

Or using a **class**:

```c
class xsd_hexBinary {
    public:
    unsigned char * _ptr;
    int _size;
};
```

When compiled by the gSOAP stub and skeleton compiler, this header file specification will generate `base64Binary` serializers and deserializers.

### 7.11 Doc/Literal XML Encoding Style

gSOAP supports doc/literal SOAP encoding of request and/or response messages. However, there are some limitations on the XML format to support (de)serialization of XML documents into C/C++ data structures. XML documents may contain constructs that gSOAP cannot parse and will simply ignore because the gSOAP XML parser has been optimized for SOAP data. This occurs when the XML document uses XML attribute values that are part of the data. Arbitrary XML documents can be (de)serialized into regular C strings or wide character strings (`wchar_t*`) by gSOAP. Because XML documents are stored in strings, an application may need a “plug-in” XML parser to decode XML content stored in strings. For details on (de)serialization XML into strings, see Section 7.11.1.

gSOAP supports doc/literal SOAP encoding either manually by setting `soap_encodingStyle`, `soap_defaultNamespace`, and `soap_disable_href`, or automatically by using a gSOAP directive in the header file. In most doc/literal cases, the `SOAP-ENV:encodingStyle` attribute needs to be absent. To do this, set `soap_encodingStyle=NULL`. Furthermore, a default namespace needs to be defined by setting `soap_defaultNamespace`. Finally, doc/literal is a limited form of serialization and does not support graphs. So setting `soap_disable_href=1`
will not produce multi-reference data. Note that cyclic data will crash the doc/literal serializer because of this setting. Also polymorphic data may cause deserialization problems due to the absence of type information in the SOAP payload (which makes us wonder why doc/literal is the default in .NET).

The LocalTimeByZipCode remote method of the LocalTime service provides the local time given a zip code and uses doc/literal SOAP encoding (using MS .NET). The following header file declares the method:

```c
int LocalTimeByZipCode(char *ZipCode, char **LocalTimeByZipCodeResult);
```

Note that none of the data types need to be namespace qualified using namespace prefixes. The use of namespace prefixes is optional for doc/literal in gSOAP. When used, the XML document will include xsi:type attributes.

To illustrate the manual doc/literal setting, the following client program sets the required properties before the call:

```c
#include "soapH.h"

int main()
{
    char *t;
    soap_encodingStyle = NULL; // don't use SOAP encoding
    soap_defaultNamespace = "http://alethea.net/webservices/"; // use the service's namespace
    soap_disable_href = 1; // don't produce multi-ref data (but can accept)
        soap_print_fault(stderr);
    else
        printf("Time = %s\n", t);
    return 0;
}
```

The SOAP request is:

```xml
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/"
```
Alternatively, the settings can be automatically set by including gSOAP directives in the header file:

```c
//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 10.1 for more details on gSOAP directives. With these directives, the gSOAP compiler generates client and server sources with the specified settings. The directives are required to produce a WSDL file for a new service that uses doc/literal encoding.

The example client program can be simplified into:

```c
#include "soapH.h"
#include "localtime.nsmap" // include generated map file
int main()
{
    char *t;
        soap_print_fault(stderr);
    else
        printf("Time = %s\n", t);
    return 0;
}
```

### 7.11.1 Serializing and Deserializing XML Into Strings

To declare a literal XML “type” to hold XML documents in regular strings, use:

```c
typedef char *XML;
```

To declare a literal XML “type” to hold XML documents in wide character strings, use:

```c
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.

Literal XML encoding should only use one input parameter and one output parameter. Here is an example of a remote method specification in which the parameters of the remote method uses literal XML encoding to pass an XML document to a service and back:

```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 `structs`. 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
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV: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:GetDocument>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Important: the literal XML encoding style MUST be specified by setting the `soap_encodingStyle` variable to NULL or a given encoding style. For example, to specify no constraints on the encoding style (which is typical) use NULL:

```c
soap_encodingStyle = NULL;
```

As a result, the `SOAP-ENV:encodingStyle` attribute will not appear in the SOAP payload.

For interoperability with Apache SOAP, use

```c
soap_encodingStyle = "http://xml.apache.org/xml-soap/literalxml";
```

The name of the response element can be changed (default is the remote method name ending with `Response`). For example:


```c
typedef char *XML;
ns__GetDocument(XML m__XMLDoc, struct ns__Document { XML m__XMLDoc; } &result);
```

8 SOAP Fault Processing

A predeclared standard SOAP Fault data structure is generated by the gSOAP stub and skeleton compiler for exchanging exception messages. This predeclared data structure is:

```c
struct SOAP_ENV__Fault
{
    char *faultcode;
    char *faultstring;
    char *faultactor;
    char *detail;
};
```

The data structure can be changed to the need of an application. To do this, include a new declaration of a `struct` SOAP_ENV__Fault or class SOAP_ENV__Fault in the header file input to the gSOAP compiler to replace the built-in data structure. For example:

```c
struct SOAP_ENV__Fault
{
    char *faultcode; // MUST be string
    char *faultstring; // MUST be string
    char *faultactor;
    char *detail; // MUST be string
    Detail *t_detail; // new detail field
};
```

where DetailType is some data type that holds application specific data such as a stack dump.

When the proxy of a remote method returns an error (see Section 6.2), the soap_fault global variable contains the SOAP Fault data.

When a remote method wants to raise an exception, it does so by assigning the global variable soap_fault with appropriate data associated with the exception and by returning the error SOAP_FAULT. For example:

```c
soap_fault.faultstring = "Stack dump";
soap_fault.detail = NULL;
soap_fault.t_detail = sp; // point to stack (needs stack serializer)
return SOAP_FAULT; // return from remote method call
```

Each remote method implementation in a service application can return a SOAP Fault upon an exception by returning an error code, see Section 4.2.1 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:

```c
int main()
{
```
some initialization code
if (initialization failed)
{
    soap_error = SOAP_FAULT;
    soap_fault.faultcode = "Server";
    soap_fault.faultstring = "Init failed";
    soap_fault.details = "...";
    soap_send_fault(); // Send SOAP Fault to client
    return 0; // Terminate
}

9 SOAP Header Processing

A predeclared standard SOAP Header data structure is generated by the gSOAP stub and skeleton compiler for exchanging SOAP messages with SOAP Headers. This predeclared data structure is:

```c
struct SOAP_ENV__Header
{
    void *dummy;
};
```

which declares an empty header (some C and C++ compilers don't accept empty structs so a transient dummy field is provided).

To adapt the data structure to a specific need for SOAP Header processing, a new `struct SOAP_ENV__Header` or `class SOAP_ENV__Header` can be added to the header file input to the gSOAP compiler.

For example, the following header can be used for transaction control:

```c
struct SOAP_ENV__Header
{
    char *t__transaction;
};
```

with client-side code:

```c
...
soap_header = NULL; // do not use a SOAP Header for the request
soap_actor = NULL; // do not use an actor (receiver is actor)
soap_call_method(...);
if (soap_header)
    cout << soap_header->t__transaction;
    // Can reset, modify, or set soap_header variable here before next call
soap_call_method(...); // 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 remote method invocation to the service. Therefore, the next request includes a transaction number:

```xml
<SOAP-ENV:Envelope ...>
```
<SOAP-ENV:Header>
  <t:transaction xsi:type="int">12345</t:transaction>
</SOAP-ENV:Header>

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:

```c
main()
{
    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();
}

int method(...)
{
    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 10.1 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 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 remote method 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 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:

```c
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:
10 Advanced Features

10.1 Customizing the WSDL and Namespace Mapping Table File Contents

A header file can be augmented with directives for the gSOAP Stub and Skeleton compiler 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 4.2.4 and 6.4). These compiler directives are specified as // -comments.

Three directives are currently supported that can be used to specify details associated with namespace prefixes used by the remote method 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).

To specify the location of a Web Service in the header file, use:

//gsoap namespace-prefix service location: 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 CGI application and the WSDL file (so URL/service-name.cgi and URL/service-name.wsdl are the actual CGI and WSDL locations).

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 affects the schema-part of the generated WSDL file and the URI in the namespace mapping table. This declaration is useful when the service declares it’s 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.
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:

```plaintext
//gsoap namespace-prefix service method-header-part: method-name header-part
```

For example:

```plaintext
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:

```plaintext
//gsoap ns service method-header-part: login transaction
//gsoap ns service method-header-part: login authentication
int ns_login(...);
```

Suppose method `ns_search` uses only the first header part (at most), then this is declared as:

```plaintext
//gsoap ns service method-header-part: search transaction
int ns_search(...);
```

Note that the method name and header part names in the directive are left unqualified.

To specify the header parts for the method input (method request message), use:

```plaintext
//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:

```plaintext
//gsoap namespace-prefix service method-output-header-part: method-name header-part
```

The declarations above only affect the WSDL. It’s the application’s responsibility to set and reset the header messages.

When doc/literal encoding is required, the service encoding can be specified in the header file:

```plaintext
//gsoap namespace-prefix service encoding: literal
```

or when the `SOAP-ENV:encodingStyle` attribute is different from the SOAP 1.1 encoding style:

```plaintext
//gsoap namespace-prefix service encoding: encoding-style
```

(Note: blanks can be used anywhere in the directive, except between // and gsoap.)

The use of these directive is best illustrated with an example. The `quotex.h` header file of the `quotex` example in the gSOAP distribution for Unix/Linux is:
The quotex example is a new Web Service created by combining two existing Web Services: the XMethods Delayed Stock Quote service and XMethods Currency Exchange service.

Namespace prefix ns3 is used for the new quotex Web Service with namespace URI urn:quotex, service name quotex, and location http://www.cs.fsu.edu/~engelen. Since the new Web Service invokes the ns1_getQuote and ns2_getRate remote methods, the service namespaces of these methods are given. The service names and locations of these methods are not given because they are only required for setting up a new Web Service for these methods (but may also be provided in the header file for documentation purposes). After invoking the gSOAP Stub and Skeleton compiler on the quotex.h header file:

```
soapcpp quotex.h
```

the WSDL of the new quotex Web Service is saved as quotex.wsdl. Since the service name (quotex), location (http://www.cs.fsu.edu/~engelen), and namespace URI (urn:quotex) were provided in the header file, the generated WSDL file does not need to be changed by hand and can be published immediately together with the compiled Web Service installed as a CGI application at the designated URL (http://www.cs.fsu.edu/~engelen/quotex.cgi and http://www.cs.fsu.edu/~engelen/quotex.wsdl).

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 %{URI}%).

### 10.2 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 the fields in a class or struct need not be (de)serialized, or the base class of a derived class should not be (de)serialized, and certain built-in classes such as ostream cannot be (de)serialized. These data types (including fields) are called “transient” and can be declared outside of gSOAP’s compilation window. Transient data type and transient fields are declared with the extern keyword or are declared within [ and ] blocks in the header file input to the gSOAP compiler. The extern keyword has a special meaning to the gSOAP 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.

First example:

```cpp
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:

```cpp
[
    class myBase  // base class need not be (de)serialized
    {
        ...
    },

class ns__myDerived : myBase
{
    ...
};
```

Third example:

```cpp
[
    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 `[ ]` 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 `[ ]` transient block.

Functions prototypes of remote methods cannot be declared transient and will result in errors when attempted.
10.3 How to Serialize Data Without XML Schema xsi:type Attributes

gSOAP serializes data in XML with xsi:type attributes when the types are declared with namespace prefixes to indicate the 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.

To omit the generation of xsi:type attributes in the serialization, simply use type declarations that do not include namespace prefixes. The only remaining issue is the (de)serialization of lists/vectors with typed elements. To declare a list/vector with typed elements, use a leading underscores for type names of the struct or class. The leading underscores in type names makes type anonymous (invisible in XML).

10.4 Function Callbacks for Customized I/O and HTTP Handling

gSOAP provides five callback functions for customized I/O and HTTP handling:
Callback (function pointer)

```c
int (*soap_fopen)(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 -1 and soap_error set to an error code
    Built-in gSOAP function: tcp_connect
```

```c
int (*soap_fpost)(const char *endpoint, const char *host, 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 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_disable_request_count≠0
    Use function soap_send(char *) to write the header contents
    Should return SOAP_OK, or a gSOAP error code
    Built-in gSOAP function: http_post
```

```c
int (*soap_fresponse)(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 6.2 and
    count is the length of the SOAP message or 0 when soap_disable_response_count≠0
    Use function soap_send(char *) to write the header contents
    Should return SOAP_OK, or a gSOAP error code
    Built-in gSOAP function: http_response
```

```c
int (*soap_fparse)()
    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(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
```

```c
int (*soap_fclose)()
    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 soap_keep_alive≠0
    Should return SOAP_OK, or a gSOAP error code
    Built-in gSOAP function: tcp_disconnect
```

```c
int (*soap_fsend)(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
```

```c
size_t (*soap_frecv)(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
```

```c
int (*soap_fignore)(const char *tag)
    Called when an unknown XML element was encountered on the input and tag is the offending XML element
    Should return SOAP_OK, or a gSOAP error code such as SOAP_MUSTUNDERSTAND to throw an exception
    Built-in gSOAP function: fignore
```

The following example uses I/O function callbacks for customized serialization of data into a buffer
and deserialization back into a datastructure:

```c
char buf[10000]; // XML buffer
int len1 = 0; // #chars written
int len2 = 0; // #chars read
// mysend: put XML in buf[]
```
int mysend(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(char *s, size_t n) {
    if (len2 + n > len1)
        n = len1 - len2;
    strncpy(s, buf + len2, n);
    len2 += n;
    return n;
}

main() {
    struct ns_person p;
    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();
    soap_enable_embedding = 1;
    soap_serialize_ns_person(&p);
    soap_put_ns_person(&p, "ns:person", NULL);
    if (soap_error)
    {
        soap_print_fault(stdout);
        exit(-1);
    }
    soap_end();
    soap_begin();
    soap_get_ns_person(&p, "ns:person", NULL);
    if (soap_error)
    {
        soap_print_fault(stdout);
        exit(-1);
    }
    soap_end();
    soap_init(); // disable callbacks
}

The soap_init() 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 remote method declaration. This struct resembles
the service request (see the generated soapStub.h file created from the header file).

The header file is:

```c
//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:

```c
#include "soapH.h"
...
int myopen(const char *endpoint, const char *host, int port)
{
    if (strncmp(endpoint, "file:", 5))
    {
        printf("File name expected\n");
        return SOAP_EOF;
    }
    if (soap_sendfd = soap_recvfd = open(host, O_RDWR, S_IWUSR—S_IRUSR)) ¡ 0)
        return SOAP_EOF;
    return SOAP_OK;
}
void myclose()
{
    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(const char *endpoint, const char *host, const char *path, const char *action, size_t count)
{
    return soap_send("Custom-generated file\n"); // writes to soap_sendfd
}
int myparse()
{
    char buf[256];
    if (lseek(soap_recvfd, 0, SEEK_SET) ¡ 0 —— soap_getline(buf, 256)) // go to begin and skip
        custom header
        return SOAP_EOF;
    return SOAP_OK;
}
main()
{
    struct ns_test r;
    struct ns_person p;
    soap_init(); // reset to default callbacks
    p.name = "John Doe";
}
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:

```c
int myignore(const char *tag)
{
    return SOAP_MUSTUNDERSTAND; // never skip elements (secure)
}
```

... soap_fignore = myignore;
soap_call_ns_.method(...); // or soap_serve();

To selectively throw an exception as soon as an unknown element is encountered but element ns:xyz can be safely ignored, use:

```c
int myignore(const char *tag)
{
    if (soap_match_tag(tag, "ns:xyz") != SOAP_OK)
        return SOAP_MUSTUNDERSTAND;
    return SOAP_OK;
}
```

... soap_fignore = myignore;
soap_call_ns_.method(...); // or soap_serve();

... struct Namespace namespaces[] =
{
    {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
    {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
    {"xsi", "http://www.w3.org/1999/XMLSchema-instance"},
    {"xsd", "http://www.w3.org/1999/XMLSchema"},
    {"ns", "some-URI"}, // the namespace of element ns:xyz
    {NULL, NULL}
};
Function `soap_match_tag` compares two tags. The second 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:

```c
struct Unknown
{
  char *tag;
  struct Unknown *next;
};
int myignore(const char *tag)
{
  char *s = (char*)soap_malloc(strlen(tag)+1);
  struct Unknown *u = (struct Unknown*)soap_malloc(sizeof(struct Unknown));
  if (s && u)
  {
    strcpy(s, tag);
    u->tag = s;
    u->next = ulist;
    ulist = u;
  }
}
```

```c
struct Unknown *ulist = NULL;
soap_fignore = myignore;
soap_call_ns_method(...); // or soap_serve()
// print the list of unknown elements
soap_end(); // clean up
```

10.5 HTTP 1.0 and 1.1

gSOAP uses HTTP 1.0 by default. gSOAP supports HTTP 1.1, but does not support all HTTP 1.1 transfer encodings such as gzipped encodings. gSOAP does support HTTP 1.1 chunked-transfer encoding. Nevertheless, the the HTTP version used can be changed by setting the global variable:

```c
soap_http_version = "1.1";
```

10.6 HTTP Keep-Alive

gSOAP supports keep-alive socket connections. To activate keep-alive support, set the global variable:

```c
soap_keep_alive = 1;
```

When a client proxy communicates with a service that closes the connection, `soap_keep_alive` will be reset to 0 afterwards.

Keep-alive support can be activated for stand-alone services:
main()
{
    ...
    soap_keep_alive = 1;
    s = soap_accept();
    ...
    while (soap_serve() == SOAP_OK && soap_keep_alive)
        ...;
...
}

The connection will be kept open on the server-side 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. If the client does not close the connection, the server will wait forever when no timeout is specified.

10.7 Timeout Management

Socket send, receive, and accept timeout values can be set to manage socket communication timeouts. The soap_send_timeout, soap_recv_timeout, and soap_accept_timeout global variables can be set to user-defined socket send and receive timeout values (measured in seconds). For example:

    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, a SOAP_EOF exception will be raised (“end of file or no input”). The soap_accept_timeout specifies the timeout value for soap_accept().

10.8 Secure SOAP Clients with HTTPS/SSL

You need to install the OpenSSL library on your platform to enable secure SOAP clients to utilize HTTPS/SSL. After installation, compile your application with option -DWITH_OPENSSL. For example on Linux:

    g++ -DWITH_OPENSSL myclient.cpp stdsoap.cpp soapC.cpp soapClient.cpp -lssl -lcrypto

or Unix:

    g++ -DWITH_OPENSSL myclient.cpp stdsoap.cpp soapC.cpp soapClient.cpp -lxnet -lsocket -lnsl
    -lssl -lcrypto
or you can add the following line to stdsoap.h:

```
#define WITH_OPENSSL
```

A client program simply uses the prefix https: instead of http: in the endpoint URL of a remote method call to a Web Service to use encrypted transfers (if the service supports HTTPS). For example:

```
soap_call_ns__mymethod("https://domain/path/secure.cgi", ", ", ...);
```

By default, server authentication is disabled. To enable server authentication, use:

```
soap_require_server_auth = 1;
```

This will force server authentication for all calls over HTTPS.

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) { }
```

### 10.9 Secure SOAP 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. Therefore, HTTPS/SSL support must be configured for the Web server (not Web Service).

SSL support for stand-alone gSOAP Web services is accomplished by calling `soap_ssl_accept` after `soap_accept`. In addition, a key file, CA file, DH file, and password need to be supplied. Instructions on how to do this can be found in the OpenSSL documentation. To enable OpenSSL, first install OpenSSL and use option `-DWITH_OPENSSL` with your C or C++ compiler, for example:

```
g++ -DWITH_OPENSSL -o myprog myprog.cpp stdsoap.cpp soapC.cpp soapServer.cpp -lssl -lcrypto
```

An example secure Web service:

```c
int main()
{
    int m, s;
    soap_keyfile = "server.pem"; // must be resident key file
    soap_cafile = "cacert.pem"; // must be resident CA file
    soap_dhfile = "dh512.pem"; // must be resident DH file
    soap_password = "password"; // password
    m = soap_bind("linprog2.cs.fsu.edu", 18000, 100);
    if (m < 0)
    {
        soap_print_fault(stderr);
        exit(-1);
    }
```
In case Web services have to verify clients, use a key file, CA file, and password in an SSL-enabled client:

```c
... soap_keyfile = "client.pem";
soap_password = "password";
soap_cafie = "cacert.pem";
if (soap_call_ns._method("https://linprog2.cs.fsu.edu:18000", ",", ...)
...```

Make sure you have signal handlers set in your service and/or client applications to catch broken connections (SIGPIPE):

```c
signal(SIGPIPE, sigpipe_handle);
```

where, for example:

```c
void sigpipe_handle(int x) { }
```

### 10.10 Client-Side Cookie Support

Client-side cookie support is optional. To enable cookie support, compile with option `-DWITH_COOKIES`, for example:

```c
g++ -DWITH_COOKIES -o myclient stdsoap.cpp soapC.cpp soapClient.cpp
```

or add the following line to stdsoap.h:

```c
#define WITH_COOKIES
```
Client-side cookie support is fully automatic. So just (re)compile stdsoap.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. So the internal cookie database is discarded when the client program terminates.

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:

```c
soap_cookie_max = 10;
```

The cookie database is a linked list pointed to by `soap.cookies` where each node is declared as:

```c
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: got cookie from client */
    short modified; /* server-side: client cookie was modified */
    struct soap_cookie *next;
};
```

### 10.11 Server-Side Cookie Support

Server-side cookie support is optional. To enable cookie support, compile with option -DWITH_COOKIES, for example:

```bash
g++ -DWITH_COOKIES -o myserver ...
```

gSOAP provides the following cookie API for server-side cookie session control:
Function

```c
struct soap_cookie *soap_set_cookie(const char *name, const char *value, const char *domain, const char *path);
```

Add a cookie to the database with name name and value value.

If successful, returns pointer to a cookie node in the linked list, or NULL otherwise.

```c
struct soap_cookie *soap_cookie(const char *name, const char *domain, const char *path);
```

Find a cookie in the database with name name and value value.

If successful, returns pointer to a cookie node in the linked list, or NULL otherwise.

```c
char *soap_cookie_value(const char *name, const char *domain, const char *path);
```

Get value of a cookie in the database with name name.

If successful, returns the string pointer to the value, or NULL otherwise.

```c
long soap_cookie_expire(const char *name, const char *domain, const char *path);
```

Get expiration value of the cookie in the database with name name (in seconds).

If successful, returns SOP_OK, or SOAP_EOF otherwise.

```c
int soap_set_cookie_expire(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).

If successful, returns SOP_OK, or SOAP_EOF otherwise.

```c
int soap_set_cookie_session(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).

If successful, returns SOP_OK, or SOAP_EOF otherwise.

```c
int soap_clr_cookie_session(const char *name, const char *domain, const char *path);
```

Clear cookie in the database with name name to be a session cookie.

If successful, returns SOP_OK, or SOAP_EOF otherwise.

```c
void soap_clr_cookie(const char *name, const char *domain, const char *path);
```

Remove cookie from the database with name name.

If successful, returns SOP_OK, or SOAP_EOF otherwise.

```c
int soap_getenv_cookies();
```

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 SOP_OK, or SOAP_EOF otherwise.

```c
void soap_free_cookies();
```

Release cookie database.

---

The following global variables are used to define the current domain and path:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>const char *soap_cookie_domain</td>
<td>MUST be set to the domain (host) of the service</td>
</tr>
<tr>
<td>const char *soap_cookie_path</td>
<td>MAY be set to the default path to the service</td>
</tr>
<tr>
<td>int soap_cookie_max</td>
<td>maximum cookie database size (default=32)</td>
</tr>
</tbody>
</table>

The `soap_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:

```c
int main()
{
```
int m, s;
soap_cookie_domain = "...";
soap_cookie_path = "/"; // the path which is used to filter/set cookies with this destination
if (argc < 2)
{
    soap_getenv_cookies(); // CGI app: grab cookies from 'HTTP_COOKIE' env var
    soapServe();
}
else
{
    m = soap_bind(NULL, atoi(argv[1]), 100);
    if (m < 0)
        exit(-1);
    for (int i = 1; i++)
    {
        s = soap_accept();
        if (s < 0)
            exit(-1);
        soapServe();
        soap_end(); // clean up
    }
}
return 0;

int ck_demo(...)
{
    int n;
    const char *s;
    s = soap_cookie_value("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("demo", s, NULL, NULL);
    soap_set_cookie_expire("demo", 5, NULL, NULL); // cookie may expire at client-side in 5 seconds
    return SOAP_OK;
}

10.12 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 to the proxy's host name and port, respectively. For example:

    soap_proxy_host = "proxyhostname";
    soap_proxy_port = 8080;
    if (soap_call_ns_.method("http://host:port/path", "action", ...))
        soap_print_fault(stderr);
    else
        ...

113
The global variables `soap_proxy_host` and `soap_proxy_port` keep their values through the remove method calls, so they only need to be set once.

### 10.13 FastCGI Support

To enable FastCGI support, install FastCGI and compile with option `-DWITH_FASTCGI` or add

```c
#define WITH_FASTCGI
```

This to `stdsoap.h`. 