gSOAP 2.3 User Guide

Robert van Engelen
Genivia inc. and Florida State University
engelen@acm.org

September 24, 2003

Contents

1 Introduction **6**

2 Notational Conventions **7**

3 Differences Between gSOAP Versions 2.1 (and Earlier) and 2.2 **8**

4 Differences Between gSOAP Versions 1.X and 2.X **8**

5 Interoperability **10**

6 Quick User Guide **11**

6.1 How to Use the gSOAP Stub and Skeleton Compiler to Build SOAP Clients . . . 12

6.1.1 Example . . . 12

6.1.2 Namespace Considerations . . . 16

6.1.3 Example . . . 17

6.1.4 How to Generate C++ Client Proxy Classes . . . 18

6.1.5 XSD Type Encoding Considerations . . . 19

6.1.6 Example . . . 20

6.1.7 How to Change the Response Element Name . . . 21

6.1.8 Example . . . 21

6.1.9 How to Specify Multiple Output Parameters . . . 22

6.1.10 Example . . . 22

6.1.11 How to Specify Output Parameters With struct/class Compound Data Types . . . 23

6.1.12 Example . . . 24

6.1.13 How to Specify Anonymous Parameter Names . . . 26

6.1.14 How to Specify a Method with No Input Parameters . . . 27

6.1.15 How to Specify a Method with No Output Parameters . . . 27

6.2 How to Use the gSOAP Stub and Skeleton Compiler to Build SOAP Web Services 28

6.2.1 Example . . . 28
6.2.2 How to Create a Stand-Alone gSOAP Service . . . 30
6.2.3 How to Create a Multi-Threaded Stand-Alone Service . . . 32
6.2.4 How to Pass Application Data to Service Methods . . . 34
6.2.5 Some Web Service Implementation Issues . . . 35
6.2.6 How to Generate C++ Server Object Classes . . . 35
6.2.7 How to Generate WSDL Service Descriptions . . . 36
6.2.8 Example . . . 37
6.2.9 How to Import WSDL Service Descriptions . . . 39
6.2.10 How to Use Client Functionalities Within a Service . . . 39
6.3 How to Use gSOAP for Asynchronous One-Way Message Passing . . . 42
6.4 How to Use the SOAP Serializers and Deserializers to Save and Load Application Data . . . 43
6.4.1 Serializing a Data Type . . . 43
6.4.2 Deserializing a Data Type . . . 47
6.4.3 Example . . . 49
6.4.4 Serializing and Deserializing Class Instances to Streams . . . 52
6.4.5 How to Specify Default Values for Omitted Data . . . 53

7 Using the gSOAP Stub and Skeleton Compiler 55
7.1 Compiler Options . . . 56
7.2 SOAP 1.1 Versus SOAP 1.2 . . . 56
7.3 The soapdefs.h Header File . . . 57
7.4 How to Build Modules and Libraries with the gSOAP #module Directive . . . 58
7.5 How to use the gSOAP #import Directive . . . 59
7.6 How to Use #include and #define Directives . . . 59
7.7 Compiling a gSOAP Client . . . 60
7.8 Compiling a gSOAP Web Service . . . 60
7.9 Using gSOAP for Creating Web Services and Clients in Pure C . . . 61
7.10 Limitations of gSOAP . . . 62
7.11 Compile Time Flags . . . 63
7.12 Run Time Flags . . . 63
7.13 Memory Management . . . 65
7.13.1 Memory Management Policies . . . 66
7.13.2 Intra-Class Memory Management . . . 68
7.14 Debugging . . . 70
7.15 Libraries . . . 70

8 The gSOAP Remote Method Specification Format 71
8.1 Remote Method Parameter Passing . . . 72
8.2 Error Codes . . . 74
8.3 C/C++ Identifier Name to XML Name Translations . . . 76
8.4 Namespace Mapping Table . . . 77

9 gSOAP Serialization and Deserialization Rules 79
9.1 Primitive Type Encoding . . . 79
9.2 How to Encode and Decode Primitive Types as XSD Types . . . 79
9.2.1 How to Use Multiple C/C++ Types for a Single Primitive XSD Type . . . 86
9.2.2 How to use Wrapper Classes to Specify Polymorphic Primitive Types . . . 86
9.2.3 XML Schema Type Decoding Rules . . . 88
9.2.4 Multi-Reference Strings . . . 91
9.2.5 "Smart String" Mixed-Content Decoding . . . 91
9.2.6 STL Strings . . . 92
9.2.7 Changing the Encoding Precision of **float** and **double** Types . . . 92
9.2.8 INF, -INF, and NaN Values of **float** and **double** Types . . . 93
9.3 Enumeration Type Encoding and Decoding . . . 93
9.3.1 Symbolic Encoding of Enumeration Constants . . . 93
9.3.2 Encoding of Enumeration Constants . . . 94
9.3.3 Initialized Enumeration Constants . . . 95
9.3.4 How to "Reuse" Symbolic Enumeration Constants . . . 95
9.3.5 Boolean Enumeration Type Encoding and Decoding for C Compilers . . . 95
9.3.6 Bitmask Enumeration Encoding and Decoding . . . 96
9.4 Struct Encoding and Decoding . . . 96
9.5 Class Instance Encoding and Decoding . . . 97
9.5.1 Example . . . 98
9.5.2 Initialized **static const** Fields . . . 99
9.5.3 Class Methods . . . 99
9.5.4 Getter and Setter Methods . . . 100
9.5.5 Streaming XML with Getter and Setter Methods . . . 101
9.5.6 Polymorphism, Derived Classes, and Dynamic Binding . . . 101
9.5.7 XML Attributes . . . 105
9.5.8 QName Attributes and Elements . . . 106
9.6 Pointer Encoding and Decoding . . . 107
9.6.1 Multi-Reference Data . . . 107
9.6.2 NULL Pointers and Nil Elements . . . 108
9.7 Void Pointers . . . 108
9.8 Fixed-Size Arrays . . . 110
9.9 Dynamic Arrays . . . 110

9.9.1 One-Dimensional Dynamic Arrays . . . 111
9.9.2 Example . . . 112
9.9.3 One-Dimensional Dynamic Arrays With Non-Zero Offset . . . 113
9.9.4 Nested One-Dimensional Dynamic Arrays . . . 114
9.9.5 Multi-Dimensional Dynamic Arrays . . . 115
9.9.6 Encoding XML Generics Containing Dynamic Arrays . . . 116
9.9.7 STL Containers . . . 117
9.9.8 Polymorphic Dynamic Arrays and Lists . . . 119
9.9.9 How to Change the Tag Names of the Elements of a SOAP Array or List 119
9.10 Base64Binary XML Schema Type Encoding . . . 120
9.11 hexBinary XML Schema Type Encoding . . . 122
9.12 Literal XML Encoding Style . . . 122
9.12.1 Serializing and Deserializing Mixed Content XML With Strings . . . 124

10 SOAP Fault Processing 126

11 SOAP Header Processing 128

12 DIME Attachment Processing 130
12.1 Non-Streaming DIME . . . 130
12.2 Streaming DIME . . . 131
12.3 Streaming Chunked DIME . . . 134

13 Advanced Features 135
13.1 Internationalization . . . 135
13.2 Customizing the WSDL and Namespace Mapping Table File Contents With gSOAP Directives . . . 135
13.3 How to Specify minOccurs and maxOccurs . . . 139
13.4 How to Specify a SimpleType and Pattern . . . 140
13.5 Transient Data Types . . . 141
13.6 Volatile Data Types . . . 142
13.7 How to Declare User-Defined Serializers and Deserializers . . . 143
13.8 How to Serialize Data Without Generating XSD Type Attributes . . . 144
13.9 Function Callbacks for Customized I/O and HTTP Handling . . . 145
13.10Speed Improvement Tips . . . 150
13.11HTTP 1.0 and 1.1 . . . 151
13.12HTTP Keep-Alive . . . 151
13.13HTTP Chunked Transfer Encoding . . . 153
13.14HTTP Buffered Sends . . . 153
13.15HTTP Authentication . . . 153
13.16HTTP Proxy Authentication . . . 154

13.17 Timeout Management for Non-Blocking Operations . . . 155
13.18 Socket Options and Flags . . . 156
13.19 Secure SOAP Clients with HTTPS/SSL . . . 156
13.20 Secure SOAP Web Services with HTTPS/SSL . . . 157
13.21 SSL Authentication Callback . . . 160
13.22 SSL Certificates . . . 160
13.23 Zlib Compressed Messages . . . 161
13.24 Client-Side Cookie Support . . . 163
13.25 Server-Side Cookie Support . . . 164
13.26 Connecting Clients Through Proxy Servers . . . 166
13.27 FastCGI Support . . . 166
13.28 How to Create gSOAP Applications With a Small Memory Footprint . . . 166
13.29 Build a Client or Server in a C++ Code Namespace . . . 167
13.30 How to Create Client/Server Libraries . . . 168
13.30.1 C++ Example . . . 169
13.30.2 C Example . . . 171
13.31 How to Create DLLs . . . 173
13.31.1 Create the Base stdsoap2.dll . . . 173
13.31.2 Creating Client and Server DLLs . . . 173
13.31.3 gSOAP Plug-ins . . . 174

1 Introduction

The gSOAP toolkit provides a unique SOAP-to-C/C++ language binding for the development of SOAP Web Services and applications that require XML processing. 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 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. The gSOAP WSDL importer "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.
- gSOAP includes a WSDL generator for convenient Web Service publishing.
- gSOAP includes a WSDL importer for automated client development.
- Generates source code for stand-alone Web Services and clients.
- Ideal for building web services that are compute-intensive and are therefore best written in C and C++.
- Platform independent: Windows, Unix, Linux, Mac OS X, Pocket PC, etc.
- Fast *in situ* serialization and deserialization with XML 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, but the SOAP 1.2 specification is in a drafting stage.
- DIME compliant attachments with streaming capabilities.
- Zlib deflate and gzip compression.

- Includes HTTP, TCP/IP, XML, and DIME stacks.
- Supports one-way messaging, including asynchronous send and receive operations.
- Supports saving and loading of XML serialized C/C++ data structures to/from files.
- The schema-specific XML pull parser is fast and efficient and does not require intermediate data storage for demarshalling to save 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 with or without DIME attachments by small-memory devices such as PDAs.
- Supports C++ single class inheritance, dynamic binding, overloading, arbitrary pointer structures such as lists, trees, graphs, cyclic graphs, fixed-size arrays, (multi-dimensional) dynamic arrays, enumerations, built-in 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 shell/batch commands.

Bold font denotes C and C++ keywords.

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

[Optional] denotes an optional construct.

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

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

Run-time options and flags have been changed to enable separate recv/send settings for transport, content encodings, and mappings. The flags are divided into four classes: transport (IO), content encoding (ENC), XML marshalling (XML), and C/C++ data mapping (C). The old-style flags soap_disable_X and soap_enable_X, where X is a particular feature, are depricated. See Section 7.12 for more details.

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

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

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

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

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

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

An environment can be reused as many times as necessary and does not need to be reinitialized in doing so. A new environment is only required for each new thread to guarantee exclusive access to a new runtime environment by each thread. For example, the following code stack-allocates the runtime environment which is used for multiple remote method calls:

```
int main()
{
  struct soap soap;
  ...
  soap_init(&soap); // initialize runtime environment
  ...
  soap_call_ns__method1(&soap, ...); // make a remote call
  ...
  soap_call_ns__method2(&soap, ...); // make another remote call
  ...
  soap_end(&soap); // clean up
  ...
}
```

The runtime environment can also be heap allocated:

```
int main()
{
  struct soap *soap;
  ...
  soap = soap_new(); // allocate and initialize runtime environment
  if (!soap) // couldn't allocate: stop
  ...
  soap_call_ns__method1(soap, ...); // make a remote call
  ...
  soap_call_ns__method2(soap, ...); // make another remote call
  ...
  soap_end(soap); // clean up
  ...
  free(soap); // deallocate runtime environment
}
```

A service need to allocate and initialize an environment before calling soap_serve:

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

```
  soap_serve(&soap);
}
```

Or alternatively:

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

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

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

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

Section 6.2.3 presents a multi-threaded stand-alone Web Service that handles multiple SOAP requests by spawning a thread for each request.

5 Interoperability

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

Apache 2.2

Apache Axis

ASP.NET

Cape Connect

Delphi

easySOAP++

eSOAP

Frontier

GLUE

Iona XMLBus

kSOAP

MS SOAP

Phalanx

SIM

SOAP::Lite

SOAP4R

Spray

SQLData

Wasp Adv.

Wasp C++

White Mesa

xSOAP

ZSI

4S4C

6 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 compiler without a detailed understanding of the SOAP protocol when gSOAP client-server 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 gSOAP Web services and clients that connect to and interoperate with other SOAP implementations such as Apache Axis, SOAP::Lite, and .NET. This requires some details of the SOAP and WSDL protocols to be understood.

6.1 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 parameters of a remote method contain elaborate data structures 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 header file can be generated from a WSDL (Web Service Description Language) documentation of a service with the gSOAP WSDL importer, see 6.2.9. 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 resulting 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 compound 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 in XML.

The remote method name and its parameterization can be found with a SOAP Web service description, typically in the form of an XML schema. For SOAP 1.1 RPC encoding, there is a one-to-one correspondence between the XML schema description of a SOAP remote method and the C/C++ type declarations required to build a client application for the Web service. There is also an almost one-to-one correspondence between the schemas and the C/C++ type declarations for SOAP literal encoding. The schemas are typically part of the WSDL specification of a SOAP Web service. The gSOAP WSDL importer converts WSDL service descriptions into header files.

6.1.1 Example

The getQuote remote method of XMethods Delayed Stock Quote service provides a delayed stock quote for a given ticker name. The WSDL description of the XMethods Delayed Stock Quote service provides the following details:

Endpoint URL:	`http://services.xmethods.net:80/soap`
SOAP action:	"" (2 quotes)
Remote method namespace:	`urn:xmethods-delayed-quotes`
Remote method name:	`getQuote`
Input parameter:	`symbol` of type `xsd:string`
Output parameter:	`Result` of type `xsd:float`

The following getQuote.h **header file** is created from the WSDL description with the WSDL importer:

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

The header file essentially specifies the service details in C/C++ with directives for the gSOAP compiler. 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 8.2 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 6.1.2. Basically, a namespace prefix is distinghuished 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 into a dash in XML, because dashes are more frequently used in XML compared to underscores, see Section 8.3.)

The gSOAP compiler is invoked from the command line with:

```
soapcpp2 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 the following function prototype generated by the gSOAP compiler:

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

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

Note that the parameters of the soap_call_ns1__getQuote function are identical to the ns1__getQuote function prototype with three additional input parameters: soap must be a valid pointer to a gSOAP runtime environment, 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 stub to retrieve the latest AOL stock quote from the XMethods Delayed Stock Quote service:

```
#include "soapH.h" // obtain the generated stub
int main()
{
  struct soap soap; // gSOAP runtime environment
  float quote;
  soap_init(&soap); // initialize runtime environment (only once)
  if (soap_call_ns1__getQuote(&soap, "http://services.xmethods.net:80/soap", "", "AOL",
quote) == SOAP_OK)
    cout << "Current AOL Stock Quote = " << quote;
  else // an error occurred
    soap_print_fault(&soap, stderr); // display the SOAP fault message on the stderr stream
  soap_end(&soap); // clean up
}
```

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 stub 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.

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

Function	**Description**
soap_init(**struct** soap *soap)	Initializes a runtime environment (required only once)
soap_init2(**struct** soap *soap, int imode, int omode)	Initializes a runtime environment and set in/out mode flags
struct soap *soap_new()	Allocates, initializes, and returns a pointer to a runtime environment
struct soap *soap_copy(**struct** soap *soap)	Allocates a new runtime invironment and copies contents of the argument environment such that the new environment does not share data with the argument environment
soap_done(**struct** soap *soap)	Reset, close communications, and remove callbacks

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

When the example client application is invoked, the SOAP request is performed by the stub routine soap_call_ns1__getQuote, which generates the following SOAP RPC 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"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/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:xsd='http://www.w3.org/1999/XMLSchema'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
<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 SOAP RPC response is parsed by the stub. 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:

```
...
struct soap soap;
float quotes[3]; char *myportfolio[] = {"IBM", "AOL", "MSDN"};
soap_init(&soap); // need to initialize only once
for (int i = 0; i < 3; i++)
  if (soap_call_ns1__getQuote(&soap, "http://services.xmethods.net:80/soap", "", myport-
folio[i], quotes[i]) != SOAP_OK)
    break;
if (soap.error) // an error occurred
  soap_print_fault(&soap, stderr);
```

```
soap_end(&soap); // clean up all deserialized data
...
```

This client composes an array of stock quotes by calling the `ns1__getQuote` stub routine 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 in the form of a WSDL document.

6.1.2 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 distinghuished 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 bound to the **namespace name** `urn:xmethods-delayed-quotes` through the `xmlns:n="urn:xmethods-delayed-quotes` binding. The use of namespace prefixes and namespace names is also required to enable SOAP applications to validate the content of SOAP messages. The namespace name in the service response is verified by the stub routine by using the information supplied in a **namespace mapping table** that is required to be part of gSOAP client and service application codes. The table is accessed at run time to resolve namespace bindings, both by the generated stub's data structure serializer for encoding the client request and by the generated stub's data structure deserializer to decode and validate the service response. The namespace mapping table should **not** be part of the header file input to the gSOAP stub and skeleton compiler. Service details including namespace bindings may be provided with gSOAP directives in a header file, see Section 13.2.

The namespace mapping table for the Delayed Stock Quote client is:

```
struct Namespace namespaces[] =
{   // {"ns-prefix", "ns-name"}
    {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"}, // MUST be first
    {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"}, // MUST be second
    {"xsi", "http://www.w3.org/2001/XMLSchema-instance"}, // MUST be third
    {"xsd", "http://www.w3.org/2001/XMLSchema"}, // 2001 XML schema
    {"ns1", "urn:xmethods-delayed-quotes"}, // given by the service description
    {NULL, NULL} // end of table
};
```

The first four namespace entries in the table consist of the standard namespaces used by the SOAP 1.1 protocol. In fact, the namespace mapping table is explicitly declared to enable a programmer to specify the SOAP encoding style and to allow the inclusion of namespace-prefix with namespace-name bindings to comply to the namespace requirements of a specific SOAP service. For example, the namespace prefix `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:

```
...
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/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.

6.1.3 Example

The incorporation of namespace prefixes into C++ identifier names is necessary to distinguish remote methods that share the same name but are provided by separate Web services and/or organizations. Consider for example:

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

Recall that the namespace prefix is always separated from the name of a remote method by a pair of underscores (__).

This example enables a client program to connect to a (hypothetical) Stock Quote service with remote methods that can only be distinghuished by their namespaces. Consequently, two different namespace prefixes had to be 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:

```
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 name of a data type by a pair of underscores (__).

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

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

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

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

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

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

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

6.1.4 How to Generate C++ Client Proxy Classes

Proxy classes for C++ client applications are automatically generated by the gSOAP compiler. To illustrate the generation of a proxy class, the getQuote.h header file example of the previous section is augmented with the appropriate directives to enable the gSOAP compiler to generate the proxy class. Similar directives are included in the header file by the WSDL importer.

```
// Content of file "getQuote.h":
//gsoap ns1 service name: Quote
//gsoap ns1 service location: http://services.xmethods.net/soap
//gsoap ns1 schema namespace: urn:xmethods-delayed-quotes
//gsoap ns1 service method-action: getQuote ""
int ns1__getQuote(char *symbol, float &Result);
```

The first three directives provide the service name which is used to name the proxy class, the service location (endpoint), and the schema. The fourth directive defines the optional SOAPAction, which is a string associated with SOAP 1.1 operations. This directive must be provided for each remote method when the SOAPAction is required. Compilation of this header file with the gSOAP compiler soapcpp2 creates a new file soapQuoteProxy.h with the following contents:

```
#include "soapH.h"
class Quote
{ public:
  struct soap *soap;
  const char *endpoint;
  Quote() { soap = soap_new(); endpoint = "http://services.xmethods.net/soap"; };
  ~Quote() { if (soap) { soap_destroy(soap); soap_end(soap); soap_done(soap); free((void*)soap);
}};
  int getQuote(char *symbol, float &Result) { return soap ? soap_call_ns1__getQuote(soap, end-
point, "", symbol, Result) : SOAP_EOM; };
};
```

The gSOAP environment and endpoint are declared public to enable access for run-time customization.

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

```
#include "soapQuoteProxy.h" // get proxy
#include "Quote.nsmap" // get namespace bindings
int main()
{
  Quote q;
  float r;
  if (q.getQuote("AOL", r) == SOAP_OK)
    std::cout << r << std::endl;
  else
    soap_print_fault(q.soap, stderr);
  return 0;
}
```

The Quote constructor allocates and initializes a gSOAP environment for the instance. All the HTTP and SOAP/XML processing is hidden and performed on the background.

You can use soapcpp2 compiler option `-n` together with `-p` to create a local namespaces table to avoid link conflicts when you need multiple namespace tables or need to combine multiple clients, see also Sections 7.1 and 13.30, and you can use a C++ code **namespace** to create a namespace qualified proxy class, see Section 13.29.

6.1.5 XSD Type Encoding Considerations

Many SOAP services require the explicit use of XML schema types in the SOAP payload. The default encoding, which is also adopted by the gSOAP compiler, assumes SOAP RPC encoding which only requires the use of types to handle polymorphic cases. Nevertheless, the use of XSD typed messages is adviced to improve interoperability. XSD types are introduced with **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 (XSD type). For example, the following **typedef** declarations define various built-in XSD types implemented as primitive C/C++ types:

```
// Contents of header file:
...
```

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

This simple mechanism informs the gSOAP compiler to generate serializers and deserializers that explicitly encode and decode the primitive C++ types as built-in primitive XSD types when the typedefed type is used in the parameter signature of a remote method (or when used nested within structs, classes, and arrays). At the same time, the use of **typedef** does not force any recoding of a C++ client or Web service application as the internal C++ types used by the application are not required to be changed (but still have to be primitive C++ types, see Section 9.2.2 for alternative class implementations of primitive XSD types which allows for the marshalling of polymorphic primitive types).

6.1.6 Example

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

```
// 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:

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

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

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

```
...
<SOAP-ENV:Body>
<ns1:getQuote><symbol xsi:type="xsd:string">AOL</symbol>
</ns1:getQuote>
</SOAP-ENV:Body>
...
```

The service response is:

```
...
<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 names (URIs) that are bound to the `xsd` namespace prefix. The stub also expects the `getQuoteResponse` element to be associated with 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 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 when the URIs do not match.

6.1.7 How to Change the Response Element Name

There is no standardized convention for the response element name in a SOAP response message, 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.

6.1.8 Example

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

```
// 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:

```
...
<SOAP-ENV:Body>
<ns1:getQuote><symbol xsi:type="xsd:string">AOL</symbol>
</ns1:getQuote>
</SOAP-ENV:Body>
...
```

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.

6.1.9 How to Specify Multiple Output Parameters

The gSOAP stub and skeleton compiler uses the convention that the **last parameter** of the function prototype declaration of a remove method in a header file is also the **only single output parameter** of the method. 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 6.1.7. 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 6.1.13.

6.1.10 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);
```

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>
<SSN>999 99 9999</SSN>
</ns3:getNames>
...
```

The response by a SOAP service could be:

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

where `first` and `last` are the output parameters of the getNames 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**:

```
// Contente 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>
<name>SOAP</name>
</X-rox:copy-name>
...
```

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

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

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

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

If the single 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 6.1.7.

6.1.12 Example

This is is best illustrated with an example. The Flighttracker service by ObjectSpace provides real time flight information for flights in the air. It requires an airline code and flight number as parameters. The 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 8.3 for details on the translation of C++ identifiers to XML element names.

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

```
struct soap soap;
...
soap_init(&soap);
...
soap_call_ns1__getFlightInfo(&soap, "testvger.objectspace.com/soap/servlet/rpcrouter",
  "urn:galdemo:flighttracker", "UAL", "184", r);
...
struct Namespace namespaces[] =
{
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
  {"SOAP-ENC","http://schemas.xmlsoap.org/soap/encoding/"},
  {"xsi", "http://www.w3.org/1999/XMLSchema-instance"},
  {"xsd", "http://www.w3.org/1999/XMLSchema"},
  {"ns1", "urn:galdemo:flighttracker"},
  {"ns2", "http://galdemo.flighttracker.com"},
  {NULL, NULL}
};
```

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

```
POST /soap/servlet/rpcrouter HTTP/1.1
Host:  testvger.objectspace.com
Content-Type:  text/xml
Content-Length:  634
SOAPAction:  "urn:galdemo:flighttracker"

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/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:

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

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/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>
<flightNumber xsi:type="xsd:string">184</flightNumber>
</return>
</ns1:getFlightInfoResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The proxy returns the service response in variable r of type **struct** ns1__getFlightInfoResponse and this information can be displayed by the client application with the following code fragment:

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

This code displays the service response as:

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

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

6.1.13 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 striclty required to match a service's view of the parameter names. Although this convention is likely to be deprecated in SOAP 1.2, the gSOAP compiler can generate stub and skeleton routines that support anonymous parameters. Parameter names are implicitly anonymous by omitting the parameter names in the function prototype of the remote method. For example:

```
// Contents of "getQuote.h":
typedef char *xsd__string;
typedef float xsd__float;
int ns1__getQuote(xsd__string, xsd__float&);
```

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

For example:

```
// Contents of "getQuote.h":
typedef char *xsd__string;
typedef float xsd__float;
int ns1__getQuote(xsd__string symbol, xsd__float &_return);
```

Or, alternatively with a response **struct**:

```
// 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.

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

6.1.14 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:

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

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

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

6.1.15 How to Specify a Method with No Output Parameters

To specify a remote method that has no output parameters, just provide a function prototype with a response struct that is empty. For example:

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

Since the response struct is empty, no output parameters are specified. Most C/C++ compilers allow empty **struct**s. For those that don't, use a **void*** parameter in the struct. This parameter is not (de)serialized.

Some SOAP resources refer to SOAP RPC with empty responses as **one way** SOAP messaging. However, we refer to one-way massaging by asynchronous explicit send and receive operations as described in Section 6.3. We found this view of one-way SOAP messaging more useful by providing a message passing alternative to SOAP RPC.

6.2 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.

6.2.1 Example

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

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

The add and sub methods are intended to add and subtract two double floating point numbers stored in input parameters a and b and should return the result of the operation in the result output parameter. The qsrt 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:

```
soapcpp2 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:

```
// Contents of file "calc.cpp":
#include "soapH.h"
#include <math.h> // for sqrt()
main()
{
  soap_serve(soap_new()); // use the remote method request dispatcher
}
// Implementation of the "add" remote method:
int ns__add(struct soap *soap, double a, double b, double &result)
{
  result = a + b;
  return SOAP_OK;
}
// Implementation of the "sub" remote method:
int ns__sub(struct soap *soap, double a, double b, double &result)
{
  result = a - b;
  return SOAP_OK;
}
// Implementation of the "sqrt" remote method:
int ns__sqrt(struct soap *soap, double a, double &result);
{
  if (a >= 0)
  {
    result = sqrt(a);
    return SOAP_OK;
  }
  else
    return soap_receiver_fault(soap, "Square root of negative number", "I can only take the square
root of a non-negative number");
}
// As always, a namespace mapping table is needed:
struct Namespace namespaces[] =
{  // {"ns-prefix", "ns-name"}
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
  {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
  {"xsi", "http://www.w3.org/1999/XMLSchema-instance"},
  {"xsd", "http://www.w3.org/1999/XMLSchema"},
  {"ns", "urn:simple-calc"}, // bind "ns" namespace prefix
  {NULL, NULL}
};
```

Note that the remote methods have an extra input parameter which is a pointer to the gSOAP runtime environment. 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. This is SOAP 1.1 specific. SOAP 1.2 requires the soap->fault->SOAP_ENV__Reason and the soap->fault->SOAP_ENV__Detail strings to be assigned. Better is to use the soap_receiver_fault function that allocates a fault struct and sets the SOAP Fault string and details regardless of the SOAP 1.1 or SOAP 1.2 version used. The soap_receiver_fault function returns SOAP_FAULT, i.e. an application-specific fault. The fault exception will be passed on to the client of this service.

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

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

The soapcpp2 compile generates a WSDL file for this service, see Section 6.2.7.

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.

6.2.2 How to Create a Stand-Alone gSOAP Service

The deployment of a Web service as a CGI application is an easy means to provide your service on the Internet. gSOAP services can also run as stand-alone services on any port by utilizing the built-in HTTP and TCP/IP stacks. The stand-alone services can be run on port 80 thereby providing Web server capabilities restricted to SOAP RPC.

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

```
int main()
{
  struct soap soap;
  int m, s; // master and slave sockets
  soap_init(&soap);
  m = soap_bind(&soap, "machine.cs.fsu.edu", 18083, 100);
  if (m < 0)
    soap_print_fault(&soap, stderr);
  else
  {
    fprintf(stderr, "Socket connection successful: master socket = %d\n", m);
    for (int i = 1; ; i++)
    {
      s = soap_accept(&soap);
      if (s < 0)
      {
        soap_print_fault(&soap, stderr);
        break;
      }
      fprintf(stderr, "%d: accepted connection from IP=%d.%d.%d.%d socket=%d", i,
        (soap.ip>>24)&0xFF, (soap.ip>>16)&0xFF, (soap.ip>>8)&0xFF, soap.ip&0xFF, s);
      soap_serve(&soap); // process RPC request
      fprintf(stderr, "request served\n");
      soap_destroy(&soap); // clean up class instances
      soap_end(&soap); // clean up everything and close socket
    }
  }
  soap_done(&soap); // close master socket
}
```

The gSOAP functions that can be used are:

Function	Description
soap_init(**struct** soap *soap)	Initializes gSOAP runtime environment (required once)
soap_bind(**struct** soap *soap, **char** *host, **int** port, **int** backlog)	Returns master socket (backlog = max. queue size for requests). When host==NULL: host is the machine on which the service runs
soap_accept(**struct** soap *soap)	Returns slave socket
soap_end(**struct** soap *soap)	Clean up deserialized data (except class instances) and temporary data
soap_free(**struct** soap *soap)	Clean up temporary data only
soap_destroy(**struct** soap *soap)	Clean up deserialized class instances (note: this function will be renamed with option -n
soap_done(**struct** soap *soap)	Reset: close master/slave sockets and remove callbacks (see Section 13.9

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

The soap.accept_timeout attribute of the gSOAP run-time environment specifies the timeout value for a non-blocking soap_accept(&soap) call. See Section 13.17 for more details on timeout management.

See Section 7.13 for more details on memory management.

A client application connects to this stand-alone service with the 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.

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

Multi-threading a Web Service is essential when the response times for handling requests by the service are (potentially) long or when keep-alive is enabled, see Section 13.12. In case of long response times, the latencies introduced by the unrelated requests may become prohibitive for a successful deployment of a stand-alone service. When HTTP keep-alive is enabled, a client may not close the socket on time, thereby preventing other clients from connecting.

gSOAP 2.0 and higher is thread safe and supports the implementation of multi-threaded stand-alone services in which a thread is used to handle a request.

Here is an example of a multi-threaded Web Service:

```
#include "soapH.h"
#include <pthread.h>
#define BACKLOG (100) // Max. request backlog
#define MAX_THR (8) // Max. threads to serve requests
int main(int argc, char **argv)
{
  struct soap soap;
  soap_init(&soap);
  if (argc < 3) // no args: assume this is a CGI application
  {
    soap_serve(&soap); // serve request, one thread, CGI style
    soap_end(&soap); // cleanup
  }
  else
  {
    struct soap *soap_thr[MAX_THR]; // each thread needs a runtime environment
    pthread_t tid[MAX_THR];
    char *host = argv[1];
    int port = atoi(argv[2]);
    int m, s, i;
    m = soap_bind(&soap, host, port, BACKLOG);
    if (m < 0)
      exit(1);
    fprintf(stderr, "Socket connection successful %d\n", m);
    for (i = 0; i < MAX_THR; i++)
      soap_thr[i] = NULL;
    for (;;)
    {
      for (i = 0; i < MAX_THR; i++)
      {
        s = soap_accept(&soap);
        if (s < 0)
          break;
        fprintf(stderr, "Thread %d accepts socket %d connection from IP %d.%d.%d.%d\n",
i, s, (soap.ip>>24)&0xFF, (soap.ip>>16)&0xFF, (soap.ip>>8)&0xFF, soap.ip&0xFF);
        if (!soap_thr[i]) // first time around
```

```
        {
          soap_thr[i] = soap_new();
          if (!soap_thr[i])
          exit(1); // could not allocate
        }
        else // recycle soap environment
        {
          pthread_join(tid[i], NULL);
          fprintf(stderr, "Thread %d completed\n", i);
          soap_end(soap_thr[i]); // deallocate data of old thread
        }
        soap_thr[i]->socket = s;
        pthread_create(&tid[i], NULL, (void*(*)(void*))soap_serve, (void*)soap_thr[i]);
      }
    }
  }
  return 0;
}
```

The example illustrates the use of threads to improve the quality of service by handling new requests in separate threads. Each thread needs a separate runtime environment. The example above requires threads to synchronize at some point, so runaway processes can be halted (not shown in the code). The next example detaches threads. No attempt is made to synchronize threads.

```
#include "soapH.h"
#include <pthread.h>
#define BACKLOG (100) // Max. request backlog
int main(int argc, char **argv)
{
  struct soap soap;
  soap_init(&soap);
  if (argc < 3) // no args: assume this is a CGI application
  {
    soap_serve(&soap); // serve request, one thread, CGI style
    soap_end(&soap); // cleanup
  }
  else
  {
    void *process_request(void*);
    struct soap *tsoap;
    pthread_t tid;
    char *host = argv[1];
    int port = atoi(argv[2]);
    int m, s;
    m = soap_bind(&soap, host, port, BACKLOG);
    if (m < 0)
      exit(1);
    fprintf(stderr, "Socket connection successful %d\n", m);
    for (;;)
    {
      s = soap_accept(&soap);
      if (s < 0)
        break;
```

```
      fprintf(stderr, "Thread %d accepts socket %d connection from IP %d.%d.%d.%d\n",
i, s, (soap.ip>>24)&0xFF, (soap.ip>>16)&0xFF, (soap.ip>>8)&0xFF, soap.ip&0xFF);
      tsoap = soap_copy(&soap); // make a safe copy
      if (!tsoap)
        break;
      pthread_create(&tid, NULL, (void*(*)(void*))process_request, (void*)tsoap);
    }
  }
  return 0;
}
void *process_request(void *soap)
{
  pthread_detach(pthread_self());
  soap_serve((struct soap*)soap);
  soap_end((struct soap*)soap);
  free(soap);
  return NULL;
}
```

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

Function	**Description**
soap_init(**struct** soap *soap)	Initializes a runtime environment (required only once)
struct soap *soap_new()	Allocates, initializes, and returns a pointer to a runtime environment
struct soap *soap_copy(**struct** soap *soap)	Allocates a new runtime invironment and copies contents of the argument environment such that the new environment does not share data with the argument environment
soap_done(**struct** soap *soap)	Reset, close communications, and remove callbacks

A new environment is required for each new thread to guarantee exclusive access to runtime environments by threads.

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

6.2.4 How to Pass Application Data to Service Methods

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

```
{ ...
  struct soap soap;
  database_handle_type database_handle;
  soap_init(&soap);    soap.user = (void*)database_handle;
  ...
  soap_serve(&soap); // call the remove method dispatcher to handle request
  ...
}
```

```
int ns__myMethod(struct soap *soap, ...)
{ ...
  fetch((database_handle_type*)soap->user);
// get data    ...
  return SOAP_OK;
}
```

Another way to pass application data around in a more organized way is accomplished with plugins, see Section 13.31.3.

6.2.5 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

- 6.1.2 for namespace considerations.
- 6.1.5 for an explanation on how to change the encoding of the primitive types.
- 6.1.7 for a discussion on how the response element format can be controlled.
- 6.1.9 for details on how to pass multiple output parameters from a remote method.
- 6.1.11 for passing complex data types as output parameters.
- 6.1.13 for anonymizing the input and output parameter names.

6.2.6 How to Generate C++ Server Object Classes

Server object classes for C++ server applications are automatically generated by the gSOAP compiler. We illustrate the generation of an object class with a calculator example.

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

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

```
#include "soapH.h"
class Calculator : public soap
{ public:
  Quote() { soap_init(this); };
```

```
  ~Quote() { soap_destroy(this); soap_end(this); soap_done(this); }};
  int serve() { return soap_serve(this); };
};
```

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

```
#include "soapCalculatorObject.h" // get server object
#include "Calculator.nsmap" // get namespace bindings
int main()
{
  Calculator c;
  return c.serve(); // calls soap_serve to serve as CGI application (using stdin/out)
}
int ns__add(double a, double b, double &result)
{
  result = a + b;
  return SOAP_OK;
}
... sub(), mul(), and div() implementations ...
```

You can use soapcpp2 compiler option `-n` together with `-p` to create a local namespaces table to avoid link conflict when you need to combine multiple tables and/or multiple servers, see also Sections 7.1 and 13.30, and you can use a C++ code **namespace** to create a namespace qualified server object class, see Section 13.29.

6.2.7 How to Generate WSDL Service Descriptions

The gSOAP stub and skeleton compiler soapcpp2 generates WSDL (Web Service Description Language) service descriptions and XML schema files when processing a header file. The 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.

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

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

This file can be incorporated in the client/service application, see Section 8.4 for details on namespace mapping tables.

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

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

6.2.8 Example

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

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="Service"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://location/Service.wsdl"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:WSDL="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
  xmlns:tns="http://location/Service.wsdl"
  xmlns:ns="http://tempuri.org">
<types>
  <schema
    xmlns="http://www.w3.org/2000/10/XMLSchema"
    targetNamespace="http://tempuri.org"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
    <complexType name="addResponse">
      <all>
        <element name="result" type="double" minOccurs="0" maxOccurs="1"/>
      </all>
      <anyAttribute namespace="##other"/>
    </complexType>
    <complexType name="subResponse">
      <all>
        <element name="result" type="double" minOccurs="0" maxOccurs="1"/>
      </all>
      <anyAttribute namespace="##other"/>
    </complexType>
```

```
    <complexType name="sqrtResponse">
      <all>
        <element name="result" type="double" minOccurs="0" maxOccurs="1"/>
      </all>
      <anyAttribute namespace="##other"/>
    </complexType>
  </schema>
</types>
<message name="addRequest">
  <part name="a" type="xsd:double"/>
  <part name="b" type="xsd:double"/>
</message>
<message name="addResponse">
  <part name="result" type="xsd:double"/>
</message>
<message name="subRequest">
  <part name="a" type="xsd:double"/>
  <part name="b" type="xsd:double"/>
</message>
<message name="subResponse">
  <part name="result" type="xsd:double"/>
</message>
<message name="sqrtRequest">
  <part name="a" type="xsd:double"/>
</message>
<message name="sqrtResponse">
  <part name="result" type="xsd:double"/>
</message>
<portType name="ServicePortType">
  <operation name="add">
    <input message="tns:addRequest"/>
    <output message="tns:addResponse"/>
  </operation>
  <operation name="sub">
    <input message="tns:subRequest"/>
    <output message="tns:subResponse"/>
  </operation>
  <operation name="sqrt">
    <input message="tns:sqrtRequest"/>
    <output message="tns:sqrtResponse"/>
  </operation>
</portType>
<binding name="ServiceBinding" type="tns:ServicePortType">
  <SOAP:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="add">
    <SOAP:operation soapAction="http://tempuri.org#add"/>
    <input>
      <SOAP:body use="encoded" namespace="http://tempuri.org"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <SOAP:body use="encoded" namespace="http://tempuri.org"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
```

```
    </output>
  </operation>
  <operation name="sub">
    <SOAP:operation soapAction="http://tempuri.org#sub"/>
    <input>
      <SOAP:body use="encoded" namespace="http://tempuri.org"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <SOAP:body use="encoded" namespace="http://tempuri.org"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
  <operation name="sqrt">
    <SOAP:operation soapAction="http://tempuri.org#sqrt"/>
    <input>
      <SOAP:body use="encoded" namespace="http://tempuri.org"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <SOAP:body use="encoded" namespace="http://tempuri.org"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
</binding>
<service name="Service">
  <port name="ServicePort" binding="tns:ServiceBinding">
    <SOAP:address location="http://location/Service.cgi"/>
  </port>
</service>
</definitions>
```

6.2.9 How to Import WSDL Service Descriptions

Note: see README.txt in the wsdlcpp directory for installation instructions for the importer.

The creation of SOAP Web Service clients from a WSDL service description is a two-step process.

First, execute java wsdlcpp file.wsdl which generates the a header file file.h and a C-source file file.c with an example client program template. Modify the client program template to your needs. Use java wsdlcpp -c file.wsdl to generate C-only code.

Second, the header file file.h is to be processed by the gSOAP compiler by executing soapcpp2 file.h. This creates the C/C++ source files to build a client application, see 6.1. In addition, this generates a client proxy object declared in soapServiceProxy.h, where Service is the name of the service defined in the WSDL. To use this object, include the soapServiceProxy.h and Service.nsmap files in your C++ client application. The Service class provides the remote Web service methods as class members.

6.2.10 How to Use Client Functionalities Within a Service

A gSOAP service may make client calls to other services from within its remove methods. This is best illustrated with an example. The following example is a more sophisticated example that

combines the functionality of two Web services into one new SOAP Web service. The service provides a currency-converted stock quote. To serve a request, the service in turn requests the stock quote and the currency-exchange rate from two XMethods services.

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:

```
// 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:

```
// Contents of file "quotex.cpp":
#include "soapH.h" // include generated proxy and SOAP support
int main(int argc, char **argv)
{
  struct soap soap;
  float q;
  soap_init(&soap);
  if (argc ¡= 2)
    soap_serve(&soap);
  else if (soap_call_ns3__getQuote(&soap, "http://www.cs.fsu.edu/~engelen/quotex.cgi",
"", argv[1], argv[2], q))
    soap_print_fault(&soap, stderr);
  else
    printf("\nCompany %s: %f (%s)\n", argv[1], q, argv[2]);
  return 0;
}
int ns3__getQuote(struct soap *soap, char *symbol, char *country, float &result)
{
  float q, r;
  int socket = soap->socket; // save socket (stand-alone service only, does not support keep-alive)
  if (soap_call_ns1__getQuote(soap, "http://services.xmethods.net/soap", "", symbol, q)
== 0 &&
    soap_call_ns2__getRate(soap, "http://services.xmethods.net/soap", NULL, "us", coun-
try, r) == 0)
  {
    result = q*r;
    soap->socket = socket;
    return SOAP_OK;
  }
  soap->socket = socket;
  return SOAP_FAULT; // pass soap fault messages on to the client of this app
}
```

```
/* Since this app is a combined client-server, it is put together with
one header file that describes all 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(struct soap *soap, char *symbol, float &result)
{ return SOAP_NO_METHOD; } // dummy: will never be called
int ns2__getRate(struct soap *soap, char *country1, char *country2, float &result)
{ return SOAP_NO_METHOD; } // dummy: will never be called

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

To compile:

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

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

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

```
quotex.cgi 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).

6.3 How to Use gSOAP for Asynchronous One-Way Message Passing

SOAP RPC client-server interaction is synchonous: the client blocks until the server responds to the request. gSOAP also supports asynchronous one-way message passing. SOAP messaging routines are declared as function prototypes, just like remote methods for SOAP RPC. However, the output parameter is a **void** type to indicate the absence of a return value.

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

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

The gSOAP stub and skeleton compiler generates the following functions in soapClient.cpp:

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

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

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

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

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

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

The skeleton routine calls the user-implemented ns__event(**struct** soap *soap, **int** eventNo) routine (note tha absence of the 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 messages:

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

6.4 How to Use the SOAP Serializers and Deserializers to Save and Load Application Data

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 or load data stored in XML format.

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

Variable	Description
int soap.socket	socket file descriptor for input and output or -1
ostream *soap.os	(C++ only) output stream used for send operations
istream *soap.is	(C++ only) input stream used for receive operations
int soap.sendfd	when soap_socket<0, this fd is used for send operations
int soap.recvfd	when soap_socket<0, thid fd is used for receive operations

The following initializing and finalizing functions can be used:

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

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

See also Section 7.12 to control the I/O buffering and content encoding such as compression and DIME encoding.

6.4.1 Serializing a Data Type

To serialize a data type to a stream, two functions should be called to prepare the data and send the data, respectively. The first function (soap_serialize) analyzes pointers and determines if multi-references are required to encode the data and if cycles are present the object graph. The second function (soap_put) produces the XML output on a stream, for example.

The soap_serialize and soap_put function names are specific to a data type. For example, soap_serialize_float(&soap, &d) is called to serialize an **float** value and soap_put_float(&soap, &d, "number", NULL) is called to output the floating point value in SOAP tagged with the name <number>. To initialize data, the soap_default function of a data type can be used. For example, soap_default_float(&soap, &d) initializes the float to 0.0. The soap_default functions are useful to initialize complex data types such as arrays, **struct**s, and **class** instances. Note that the soap_default functions do not need the gSOAP runtime environment as a first parameter.

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

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

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

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

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

```
soap_serialize_PointerTons__Person(&soap, &p);
```

The **address of** p is passed, so the serializer can determine whether p was already serialized and to discover 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):

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

This produces:

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

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

soap_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.

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

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

If you want to serialize more than one data structure to a stream and parts of those data structures are shared through pointers, then the soap_serialize functions MUST to be called first on all of the data structures before you call 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:

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

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.

To preserve the exact structure of the data and create XML with one root, use the SOAP_XML_GRAPH output-mode flag to serialize the data in XML (see Section 7.12) to serialize multi-referenced data embedded in the structure which assures the preservation of structure but is not SOAP 1.1 compliant. To save the data as an XML tree (with one root) use the SOAP_XML_TREE flag.

For serializing class instances, method invocations MUST be used instead of function calls, for

example obj.soap_serialize(&soap) and obj.soap_put(&soap, "elt", "type"). This ensures that the proper serializers are used for serializing instances of derived classes.

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

```
struct soap soap;
myClass obj;
soap_init(&soap); // initialize
soap_begin(&soap); // start new (de)serialization phase
soap_set_omode(&soap, SOAP_XML_GRAPH);
obj.serialize(&soap);
soap.os = cout; // send to cout
soap_begin_send(&soap);
obj.put(&soap, "[namespace-prefix:]element-name1", "[namespace-prefix:]type-name1");
...
soap_end_send(&soap); // flush
soap_end(&soap); // remove temporary data structures after phase
soap_done(&soap); // finalize last use of this environment
```

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

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

Of course, when you construct an instance you must set its soap struct to a valid environment. Deserialized class instances with a soap struct data member will have their soap structs set automatically, see Section 7.13.2.

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 in XML.

Consider the following **struct**:

```
// Contents of file "tricky.h":
struct Tricky
{
```

```
  int *p;
  int n;
  int *q;
};
```

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

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

The resulting output is:

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

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

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).

6.4.2 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;
struct soap soap;
...
soap_init(&soap); // initialize at least once
[soap_imode(&soap, flags);] // set input-mode flags
soap_begin(&soap); // begin new decoding phase
[soap.is = an_input_stream;] // C++
[soap.recvfd = an_input_file_desriptpr;] // C
soap_begin_recv(&soap); // if HTTP header is present, parse it
if (!soap_get_Type1(&soap, &var1, "[namespace-prefix:]element-name1", "[namespace-prefix:]type-
name1"))
```

```
  ... error ...
if (!soap_get_Type2(&soap, &var2, "[namespace-prefix:]element-name2", "[namespace-prefix:]type-
name1"))
  ... error ...
...
soap_end_recv(&soap); // check consistancy of id/hrefs
soap_destroy(&soap); // remove deserialized class instances
soap_end(&soap); // remove temporary data, including the decoded data on the heap
soap_done(&soap); // finalize last use of the environment
```

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

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

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

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

You can deserialize class instances from a stream as follows:

```
myClass obj;
struct soap soap;
soap_init(&soap); // initialize
soap_begin(&soap); // begin new decoding phase
soap.is = cin; // read from cin
soap_begin_recv(&soap); // if HTTP header is present, parse it
if (!obj.get(&soap, "myClass", NULL)
  ... error ...
soap_end_recv(&soap); // check consistancy of id/hrefs
...
soap_destroy(&soap); // remove deserialized class instances
soap_end(&soap); // remove temporary data, including the decoded data on the heap
soap_done(&soap); // finalize last use of the environment
```

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

```
istream &operator>>(istream &i, myClass &e)
{
  if (!e.soap)
  ... error: need soap struct to deserialize (could use global struct)...
```

```
  istream *is = e.soap-¿is;
  e.soap->is = &i;
  if (soap_begin_recv(e.soap) || e.in(e.soap, NULL, NULL) || soap_end_recv(e.soap))
    ... error ...
  e.soap->is = is;
  return i;
}
```

6.4.3 Example

As an example, consider the following data type declarations:

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

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

```
// Contents of file "person.cpp":
#include "soapH.h"
int main()
{
  struct soap soap;
  ns__Person mother, father, john;
  mother.name = "Mary";
  mother.gender = female;
  mother.address.street = "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_init(&soap);
  soap_omode(&soap, SOAP_ENC_ZLIB|SOAP_XML_GRAPH); // see 7.12
  soap_begin(&soap);
  soap_begin_send(&soap);
  john.soap_serialize(&soap);
  john.soap_put(&soap, "johnnie", NULL);
  soap_end_send(&soap);
  soap_end(&soap);
  soap_done(&soap);
}
struct Namespace namespaces[] =
{
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
  {"SOAP-ENC","http://schemas.xmlsoap.org/soap/encoding/"},
  {"xsi", "http://www.w3.org/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:

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

(Depending on your system configuration, the libraries libsocket.a, libxnet.a, libnsl.a are required. Compiling on Linux typically does not require the inclusion of those libraries.) See 13.23 for details on compression with gSOAP.

Running the person application results in the compressed XML output:

```
<johnnie xsi:type="ns:Person" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/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>
</mother>
<father xsi:type="ns:Person">
<name xsi:type="xsd:Name">Stuart</name>
<gender xsi:type="ns:Gender">male</gender>
<address xsi:type="ns:Address">
<street xsi:type="xsd:string">Main st.</street>
<number xsi:type="unsignedInt">13</number>
<city href="#4"/>
</address>
</father>
</johnnie>
```

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

```
#include "soapH.h"
int main()
{
  struct soap soap;
  ns__Person *mother, *father, *john = NULL;
  soap_init(&soap);
  soap_imode(&soap, SOAP_ENC_ZLIB); // optional: gzip is detected automatically
  soap_begin(&soap);
  soap_begin_recv(&soap);
  if (soap_get_ns__Person(&soap, john, "johnnie", NULL))
    ... error ...
  mother = john->mother;
  father = john->father;
  ...
  soap_end_recv(&soap);
  soap_free(&soap); // Clean up temporary data but keep deserialized data
}
struct Namespace namespaces[] =
{
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
  {"SOAP-ENC","http://schemas.xmlsoap.org/soap/encoding/"},
  {"xsi", "http://www.w3.org/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:

```
john = soap_get_ns__Person(&soap, 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 XML 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:

```
#include "soapH.h"
main()
{
  struct soap soap;
  ns__Person *mother, *father, john;
  soap_init(&soap);
  soap_imode(&soap, SOAP_ENC_ZLIB); // optional
  soap_begin(&soap);
  soap_begin_recv(&soap);
  soap_default_ns__Person(&soap, &john);
  if (soap_get_ns__Person(&soap, &john, "johnnie", NULL))
    ... error ...
  ...
}
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.

6.4.4 Serializing and Deserializing Class Instances to Streams

C++ applications can define appropriate stream operations on objects for (de)serialization of objects on streams. This is best illustrated with an example. Consider the class

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

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

```
soapcpp2 person.h
```

gSOAP generates the (de)serializers and an instantiation function for the class soap_new_ns__person(**struct** soap *soap, **int** array) to instantiate one or more objects and associate them with a gSOAP environment. The array parameter should be -1 to instantiate one object or should be the number of objects to instantiate as an array of objects.

```
#include "soapH.h"
#include "ns.nsmap"
...
struct soap *soap = soap_new();
ns__person *p = soap_new_ns__person(soap, -1);
...
cout ¡¡ p; // serialize p in XML
...
in ¿¿ p; // parse XML and deserialize p
...
soap_destroy(soap); // deletes p too
soap_end(soap);
soap_done(soap);
```

The stream operations are implemented as follows

```
ostream &operator<<(ostream &o, const ns__person &p)
{
  if (!p-¿soap)
    return o; // need a gSOAP environment to serialize
  p.soap->os = &o;
  soap_omode(p.soap, SOAP_XML_GRAPH); // XML tree or graph
  p-¿soap_serialize(p.soap);
  soap_begin_send(p.soap);
  if (p-¿soap_put(p.soap, "person", NULL)
    || soap_end_send(p.soap))
    ; // handle I/O error
  return o;
}
istream &operator>>(istream &i, ns__person &p)
{
  if (!p-¿soap)
    return o; // need a gSOAP environment to parse XML and deserialize
  p.soap->is = &i;
  if (soap_begin_recv(p.soap)
    || p-¿soap_in(p.soap, NULL, NULL)
    || soap_end_recv(p.soap))
    ; // handle I/O error
  return i;
}
```

6.4.5 How to Specify Default Values for 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 stdsoap2.h file:

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

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

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

```
struct MyRecord
{
  char *name = "Unknown";
  int value = 9999;
  enum Status { active, passive } status = passive;
}
```

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

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

```
int ns__login(char *username = "anonymous", char *password = "guest", bool granted);
```

When the request message lacks username and password parameters, e.g.:

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

```
  <SOAP-ENV:Body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <ns:login>
    </ns:login>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

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

7 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:

soapcpp2 [aheaderfile.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:

File Name	Description
soapH.h	Main header file to be included by all client and service sources
soapC.cpp	Serializers and deserializers for the specified data structures
soapClient.cpp	Client stub routines and proxies for all remote methods
soapServer.cpp	Service skeleton routines
soapClientLib.cpp	Client stubs combined with local static (de)serializers
soapServerLib.cpp	Service skeletons combined with local static (de)serializers
soapStub.h	A modified header file produced from the compiler input header file
.xsd	An ns.xsd file is generated with an XML schema for each namespace prefix ns used by a data structure in the header file input to the compiler, see Section 6.2.7
.wsdl	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 6.2.7
.xml	Several SOAP/XML request and response files are generated. These are example message files are valid provided that sufficient schema namespace directives are added to the header file or the generated .nsmap namespace table for the client/service is not modified by hand
.nsmap	A ns.nsmap file is generated for each namespace prefix ns used by a remote method in the header file input to the compiler, see Section 6.2.7. The file contains a namespace mapping table that can be used in the client/service sources

Both client and service applications are developed from a header file that specifies the 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 soapClientLib.cpp and soapServerLib.cpp can be used to build (dynamic) client and server libraries. The serialization routines are local (static) to avoid link symbol conflicts. You must create a separate library for SOAP Header and Fault handling, as described in Section 13.30.

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

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

7.1 Compiler Options

The compiler supports the following options:

Option	Description
-1	Use SOAP 1.1 namespaces and encodings (default)
-2	Use SOAP 1.2 namespaces and encodings
-h	Print a brief usage message
-c	Save files using extension .c instead of .cpp
-i	Interpret #include and #define directives
-m	Generate code that requires array/binary classes to explicitly free malloced array (to be depricated since soap_unlink should be used to keep specific data)
-I <path>	Use <path> (and current dir) for #import
-d <path>	Save sources in directory specified by <path>
-p <name>	Save sources with file name prefix <name> instead of "soap"
-n	When used with -p, enables multi-client and multi-server builds: Sets compiler option WITH_NONAMESPACES, see Section 7.11 Saves the namespace mapping table with name <name>_namespaces instead of namespaces Renames soap_serve() into <name>_serve() and soap_destroy() into <name>_destroy()

For example

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

Saves the sources:

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

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

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

7.2 SOAP 1.1 Versus SOAP 1.2

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

```
struct Namespace namespaces[] =
{
  {"SOAP-ENV", "http://www.w3.org/2002/06/soap-envelope"},
  {"SOAP-ENC", "http://www.w3.org/2002/06/soap-encoding"},
  {"xsi", "http://www.w3.org/2001/XMLSchema-instance", "http://www.w3.org/*/XMLSchema-
instance"},
  {"xsd", "http://www.w3.org/2001/XMLSchema", "http://www.w3.org/*/XMLSchema"},    ...
}
```

gSOAP Web services and clients can automatically switch from SOAP 1.1 to SOAP 1.2 by providing the SOAP 1.2 namespace as a pattern in the third column of a namespace table:

```
struct Namespace namespaces[] =
{
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/", "http://www.w3.org/2002/06/soap-
encoding"},
  {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/", "http://www.w3.org/2002/06/soap-
envelope"},
  {"xsi", "http://www.w3.org/2001/XMLSchema-instance", "http://www.w3.org/*/XMLSchema-
instance"},
  {"xsd", "http://www.w3.org/2001/XMLSchema", "http://www.w3.org/*/XMLSchema"},    ...
}
```

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

The gSOAP soapcpp2 compiler generates a .nsmap file with SOAP-ENV and SOAP-ENC namespace patterns similar to the above. Since clients issue a send first, they will always use SOAP 1.1 for requests when the namespace table is similar as shown above. Clients can accept SOAP 1.2 responses by inspecting the response message. To restrict gSOAP services and clients to SOAP 1.2 and to generate SOAP 1.2 service WSDLs, use soapcpp2 compiler option -2 to generate SOAP 1.2 conformant .nsmap and .wsdl files.

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

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

7.3 The soapdefs.h Header File

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

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

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

```
// Contents of soapdefs.h
#include <ostream>
#define SOAP_BUFLEN 20480 // use large send/recv buffer
```

The following header file can now refer to ostream:

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

See also Section 13.5.

7.4 How to Build Modules and Libraries with the gSOAP #module Directive

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

```
#module "name"
```

where *name* is a unique name for the module. The name is case insensitive and MUST not exceed 4 characters in length. The rest of the content of the header file should include type declarations and optionally the declrations of remote methods and SOAP Headers/Faults. When the gSOAP compiler processes the header file module, it will generate the source codes for a library. The Web services application that uses the library should use a header file that imports the module with the #import directive.

For example:

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

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

A module MUST be imported into another header file to use it and you cannot use a module alone to build a SOAP or XML application. That is, the top most header file in the import tree SHOULD NOT be a module.

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

7.5 How to use the gSOAP #import Directive

The #import directive is used to include gSOAP header files into other gSOAP header files for processing with the gSOAP compiler soapcpp2. The C #include directive cannot be used to include gSOAP header files. The #include directive is reserved to control the post-gSOAP compilation process, see 7.6.

The #import directive is used for two purposes: you can use it to include the contents of a header file into another header file and you can use it to import a module, see 7.4.

An example of the #import directive:

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

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

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

7.6 How to Use #include and #define Directives

The #include and #define directives are normally ignored by the gSOAP compiler. The use of the directives is enabled with the -i option of the gSOAP compiler, see Section 7.1. However, the gSOAP compiler will not actually parse the contents of the header files provided by the #include directives in a header file. Instead, the #include and #define directives will be added to the generated soapH.h header file **before** any other header file is included. Therefore, #include and #define directives can be used to control the C/C++ compilation process of the sources of an application.

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

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

This example also uses #define directives for various settings.

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

7.7 Compiling a gSOAP 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 stdsoap2.cpp soapC.cpp soapClient.cpp
```

Or on a Unix machine:

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

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

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

```
// Contents of file "myclient.cpp"
#include "soapH.h";
...
// A 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 6.2.7.

7.8 Compiling a gSOAP 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 stdsoap2.cpp soapC.cpp soapServer.cpp
```

Or on a Unix machine:

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

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

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

```
// Contents of file "myserver.cpp"
#include "soapH.h";
int main()
{
  soap_serve(soap_new());
}
...
// Implementations of the 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 gSOAP service is compiled and installed as a CGI application, the soap_serve function acts as a service dispatcher. It listens to standard input and invokes the method via a skeleton routine to serve a SOAP client request. After the request is served, the response is encoded in SOAP and send to standard output. The method must be implemented in the server application and the type signature of the method must be identical to the 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/C++ function.

7.9 Using gSOAP for Creating Web Services and Clients in Pure C

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

```
soapcpp2 -c quote.h
gcc -o quote quote.c stdsoap2.c soapC.c soapClient.c
```

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

7.10 Limitations of gSOAP

gSOAP is fully SOAP 1.1 (and mostly SOAP 1.2) compliant and supports all SOAP RPC and most SOAP LIT 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.

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

STL and STL templates The gSOAP compiler does not yet fully support STL. It supports STL strings std::string and std::wstring (see Section 9.2.6) and the STL containers std::deque, std::list, std::vector, and std::set, (see Section 9.9.7).

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

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

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

Directives Directives and pragmas such as #include and #define are interpreted by the gSOAP compiler. However, the interpretation is different compared to the usual handling of directives, see Section 7.6. If necessary, a traditional C++ preprocessor can be used for the interpretation of directives. For example, Unix and Linux users can use "`cpp -B`" to expand the header file, e.g. `cpp -B myfile.h | soapcpp2`. Use the gSOAP #import directive to import gSOAP header files, see 7.5.

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

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. However, a struct or class with a **void*** field can be augmented to support the (de)serialization of the **void*** as described in Section 9.7.

Pointers to sequences of elements in memory Any pointer, except for C strings which are pointers to a sequence of characters, are treated by the compiler as if the pointer points to **only one element in memory** at run-time. Consequently, the encoding and decoding routines will ignore any subsequent elements that follow the first in memory. For the same reason, arrays of undetermined length, e.g. **float** a[] cannot be used. gSOAP supports dynamic arrays using a special type convention, see Section 9.9.

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 9.9.

7.11 Compile Time Flags

The following macros (#defines) can be used to enable certain optional features:

Macro	Description
WITH_SOAPDEFS_H	includes the soapdefs.h file for custom settings, see Section 7.3
WITH_COOKIES	enables HTTP cookies, see Sections 13.24 13.25
WITH_OPENSSL	enables OpenSSL, see Sections 13.19 13.20
WITH_FASTCGI	enables FastCGI, see Sections 13.27
WITH_GZIP	enables gzip and deflate compression, see Section 13.23
WITH_ZLIB	enables deflate compression only, see Section 13.23
WITH_LEAN	creates a small-footprint executable, see Section 13.28
WITH_LEANER	creates an even smaller-footprint executable, see Section 13.28
WITH_NONAMESPACES	omit initialization of soap struct with global namespaces table and you MUST explicitly initialize soap.namespaces to point to a table see also Section 8.4
WITH_NOGLOBAL	omit SOAP Header and Fault serialization code
WITH_CASEINSENSITIVETAGS	enable case insensitive XML parsing

Caution: it is important that the macros MUST be consistently defined to compile the sources, such as stdsoap2.cpp, soapC.cpp, soapClient.cpp, soapServer.cpp, and all application sources that include stdsoap2.h or soapH.h. If the macros are not consistently used, the application will crash due to a mismatches in the declaration and access of the gSOAP environment.

7.12 Run Time Flags

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

Although gSOAP is fully SOAP 1.1 compliant, some SOAP implementations may have trouble accepting multi-reference data and/or require explicit nil data so these flags can be used to put

gSOAP in "safe mode". In addition, the embedding (or inlining) of multi-reference data is adopted in the SOAP 1.2 specification, which gSOAP automatically supports when handling with SOAP 1.2 messages. The flags are:

Flag	Description
SOAP_IO_FLUSH	Disable buffering and flush output (default for all file-based output)
SOAP_IO_BUFFER	Enable buffering (default for all socket-oriented connections)
SOAP_IO_STORE	Store entire message to calculate HTTP content length
SOAP_IO_CHUNK	Use HTTP chunking
SOAP_IO_LENGTH	Require apriori calculation of content length (this is automatic)
SOAP_IO_KEEPALIVE	Attempt to keep socket connections alive (open)
SOAP_ENC_XML	Use plain XML encoding without HTTP headers
SOAP_ENC_DIME	Use DIME encoding (automatic when DIME attachments are used)
SOAP_ENC_SSL	Encrypt encoding with SSL (automatic with "https:" endpoints)
SOAP_ENC_ZLIB	Compress encoding with Zlib (deflate or gzip format)
SOAP_XML_CANONICAL	Produce canonical XML output
SOAP_XML_STRICT	XML strict validation (fault on unknown XML elements rather than omitting them)
SOAP_XML_TREE	Serialize data as XML trees (no multi-ref, duplicate data when necessary)
SOAP_XML_GRAPH	Serialize data as an XML graph with inline multi-ref (SOAP 1.2 default)
SOAP_XML_NIL	Serialize NULL data as XML nil elements (omit by default)
SOAP_C_NOIOB	Do not fault with SOAP_IOB
SOAP_C_UTFSTRING	(De)serialize 8-bit strings "as is" (strings MUST have UTF-8 encoded content)

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

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

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

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

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

For example

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

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

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

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

7.13 Memory Management

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

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

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 functions to deallocate data collectively or individually.

The deallocation functions are:

Function Call	**Description**
soap_end(**struct** soap *soap)	Remove temporary data and deserialized data except class instances
soap_free(**struct** soap *soap)	Remove temporary data only
soap_destroy(**struct** soap *soap)	Remove all dynamically allocated class instances. Need to be called before soap_end()
soap_dealloc(**struct** soap *soap, **void** *p)	Remove malloced data at p. When p==NULL: remove all dynamically allocated (deserialized) data except class instances
soap_delete(**struct** soap *soap, **void** *p)	Remove class instance at p. When p==NULL: remove all dynamically allocated (deserialized) class instances (this is identical to calling soap_destroy(**struct** soap *soap))
soap_unlink(**struct** soap *soap, **void** *p)	Unlink data/object at p from gSOAP's deallocation chain so gSOAP won't deallocate it
soap_done(**struct** soap *soap)	Reset: close master/slave sockets and remove callbacks (see Section 13.9

Temporary data (i.e. the hash tables) are automatically removed with calls to the soap_free function which is made within soap_end and soap_done or when the next call to a stub or skeleton routine is made to send a message or receive a message. Deallocation of non-class based data is straightforward: soap_end removes all dynamically allocated deserialized data (data allocated with soap_malloc.

That is, when the client/service application does not use any class instances that are (de)marshalled, but uses structs, arrays, etc., then calling the soap_end function is safe to remove all deserialized data. The function can be called after processing the deserialized data of a remote method call or after a number of remote method calls have been made. The function is also typically called after soap_serve, when the service finished sending the response to a client and the deserialized client request data can be removed.

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

7.13.1 Memory Management Policies

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

1. the program code deletes the class instances and the class destructors in turn SHOULD delete and free any dynamically allocated data (deep deallocation) without calling the soap_end and soap_destroy functions,
2. or the class destructors SHOULD NOT deallocate any data and the soap_end and soap_destroy functions can be called to remove the data.
3. or the class destructors SHOULD mark their own deallocation and mark the deallocation of any other data deallocated by it's destructors by calling the soap_unlink function. This allows soap_destroy and soap_end to remove the remaining instances and data without causing duplicate deallocations.

With the -m option of soapcpp2 enabled (to be depricated), there is one exception which requires explicit deallocation of malloced data in the destructors of classes for array binary types:

- A dynamic array class with non-class elements 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.

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

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

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

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

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

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

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

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

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

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

Function Call	**Description**
void *soap_malloc(**struct** soap *soap, size_t n)	return pointer to n bytes
Class *soap_new_Class(**struct** soap *soap, **int** n)	instantiate n Class objects

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. Objects instantiated with soap_new_X(**struct** soap*) are removed altogether with soap_destroy(**struct**soap*). Individual objects instantiated with soap_new_X are removed with soap_delete_X(**struct** soap*, X*). For example, the following service uses temporary data in the remote method implementation:

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

An example remote method that allocates a temporary string is:

```
int ns__itoa(struct soap *soap, int i, char **a)
{
  *a = (char*)soap_malloc(soap, 11);
  sprintf(*a, "%d", i);
  return SOAP_OK;
}
```

This temporary allocation can also be used to allocate strings for the SOAP Fault data structure. For example:

```
int ns__mymethod(...)
{ ...
  if (exception)
  {
    char *msg = (char*)soap_malloc(soap, 1024); // allocate temporary space for detailed message
    sprintf(msg, "...", ...); // produce the detailed message
    return soap_receiver_fault(soap, "An exception occurred", msg); // return the server-side fault
  }
  ...
}
```

Use soap_receiver_fault(**struct** soap *soap, **const char** *faultstring, **const char** *detail) to set a SOAP 1.1/1.2 fault at the server-side. Use soap_sender_fault(**struct** soap *soap, **const char** *faultstring, **const char** *detail) to set a SOAP 1.1/1.2 unrecoverable Bad Request fault at the server-side.

gSOAP provides a function to duplicate a string into gSOAP's memory space:

```
char *soap_strdup(struct soap *soap, const char *s)
```

The function allocates space for s with soap_malloc, copies the string, and returns a pointer to the duplicated string. When s is NULL, the function does not allocate and copy the string and returns NULL.

7.13.2 Intra-Class Memory Management

When a class declaration has a **struct** soap * field, this field will be set to point to the current gSOAP run-time environment by gSOAP's deserializers and by the soap_new_Class functions. This simplifies memory management for class instances. The **struct** soap* pointer is implicitly set by the gSOAP deserializer for the class or explicitly by calling the soap_new_X function for class X. For example:

```
class Sample
{ public:
  struct soap *soap; // reference to gSOAP's run-time
  ...
  Sample();
  ~Sample();
};
```

The constructor and destructor for class Sample are:

```
Sample::Sample()
{ this->soap = NULL;
}
Sample::~Sample()
{ soap_unlink(this->soap, this);
}
```

The soap_unlink() call removes the object from gSOAP's deallocation chain. In that way, soap_destroy can be safely called to remove all class instances. The following code illustrates the explicit creation of a Sample object and cleanup:

```
struct soap *soap = soap_new(); // new gSOAP runtime
Sample *obj = soap_new_Sample(soap, -1); // new Sample object with obj->soap set to runtime
...
delete obj; // also calls soap_unlink to remove obj from the deallocation chain
soap_destroy(soap); // deallocate all (other) class instances
soap_end(soap); // clean up
```

Here is another example:

```
class ns__myClass
{ ...
  struct soap *soap; // set by soap_new_ns__myClass()
  char *name;
  void setName(const char *s);
  ...
};
```

Calls to soap_new_ns__myClass(soap, n) will set the soap field in the class instance to the current gSOAP environment. Because the deserializers invoke the soap_new functions, the soap field of the ns__myClass instances are set as well. This mechanism is convenient when Web Service methods need to return objects that are instantiated in the methods. For example

```
int ns__myMethod(struct soap *soap, ...)
{
  ns__myClass *p = soap_new_ns__myClass(soap, -1);
  p-¿setName("SOAP");
  return SOAP_OK;
}
void ns__myClass::ns__setName(const char *s)
{
  if (soap)
    name = (char*)soap_malloc(soap, strlen(s)+1);
  else
    name = (char*)malloc(strlen(s)+1);
  strcpy(name, s);
}
ns__myClass::ns__myClass()
{
  soap = NULL;
  name = NULL;
}
ns__myClass::~ns__myClass()
{
  if (!soap && name) free(name);
  soap_unlink(soap, this);
}
```

Calling soap_destroy right after soap_serve in the Web Service will destroy all dynamically allocated class instances.

7.14 Debugging

To activate message logging for debugging, un-comment the #define DEBUG directive in stdsoap2.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:

File	Description
SENT.log	The SOAP content transmitted by the application
RECV.log	The SOAP content received by the application
TEST.log	A log containing various activities performed by the application

Caution: The client and server applications may run slow due to the logging activity.

Caution: When installing a CGI application on the Web with debugging activated, the log files may sometimes not be created due to file access permission restrictions imposed on CGI applications. To get around this, create empty log files with universal write permissions. Be careful about the security implication of this.

You can test a service CGI application without deploying it on the Web. To do this, create a client application for the service and activate message logging by this client. Remove any old SENT.log file and run the client (which connects to the Web service or to another dummy, but valid address) and copy the SENT.log file to another file, e.g. SENT.tst. Then redirect the SENT.tst file to the service CGI application. For example,

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

This should display the service response on the terminal.

The file names of the log files and the logging activity can be controlled at the application level. This allows the creation of separate log files by separate services, clients, and threads. For example, the following service logs all SOAP messages (but no debug messages) in separate directories:

```
struct soap soap;
soap_init(&soap);
...
soap_set_recv_logfile(&soap, "logs/recv/service12.log"); // append all messages received in /logs/recv/service12.log
soap_set_sent_logfile(&soap, "logs/sent/service12.log"); // append all messages sent in /logs/sent/service12.log
soap_set_test_logfile(&soap, NULL); // no file name: do not save debug messages
...
soap_serve(&soap);
...
```

Likewise, messages can be logged for individual client-side remote method calls.

7.15 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 stdsoap2.cpp soapC.cpp soapClient.cpp -lsocket -lxnet -lnsl
```

These library loading options are not required with Linux.

- The gSOAP runtime uses the math library for the `NaN`, `INF`, and `-INF` floating point representations. The library is not strictly necessary and the ¡math.h¿ header file import can be commented out from the stdsoap2.h header file. The application can be linked without the -lm math library e.g. under Sun Solaris:

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

8 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 8.2. 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 8.3)

method_name it the remote method name (see identifier translation rules 8.3)

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 8.1 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 8.3. (The optional parts are shown enclosed in [].)

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

```
<[namespace-prefix:]method-nameResponse xsi:type="[namespace-prefix:]method-nameResponse>
<outparam-name xsi:type="...">...</outparam-name>
</[namespace-prefix:]method-nameResponse>
```

where the `outparam-name` accessor is the element-name representation of the outparam parameter name declaration, see Section 8.3. By convention, the response element name is the method name ending in `Response`. See 8.1 on how to change the declaration if the service response element name is different.

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

```
int soap_call_[namespace_prefix_ _]method_name(struct soap *soap, char *URL, char *action, [inparam1,
inparam2, ...,] outparam);
```

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

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

```
int soap_serve_[namespace_prefix_ _]method_name(struct soap *soap);
```

The skeleton routine, when called by a service application, will attempt to serve a request on the standard input. If no request is present or if the request does not match the method name, SOAP_NO_METHOD is returned. The skeleton routines are automatically called by the generated soap_serve routine that handles all requests.

8.1 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 by a pointer to the data. Of course, passing a pointer to the data is prefered 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 7.10

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 6.1.7. Hence, if the output parameter has to be a **struct** or **class**, a response **struct** or **class** MUST be declared as well. In addition, if a 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 8.3)

response_element_name it the name of the response element (see identifier translation rules 8.3)

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 8.3. (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 8.3. (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 6.1.13.

8.2 Error Codes

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

#	Code	Description
0	SOAP_OK	No error
1	SOAP_CLI_FAULT*	The service returned a client fault (SOAP 1.2 Sender fault)
2	SOAP_SVR_FAULT*	The service returned a server fault (SOAP 1.2 Receiver fault)
3	SOAP_TAG_MISMATCH	An XML element didn't correspond to anything expected
4	SOAP_TYPE_MISMATCH	An XML schema type mismatch
5	SOAP_SYNTAX_ERROR	An XML syntax error occurred on the input
6	SOAP_NO_TAG	Begin of an element expected, but not found
7	SOAP_IOB	Array index out of bounds
8	SOAP_MUSTUNDERSTAND*	An element needs to be ignored that need to be understood
9	SOAP_NAMESPACE	Namespace name mismatch (validation error)
10	SOAP_OBJ_MISMATCH	Mismatch in the size and/or shape of an object
11	SOAP_FATAL_ERROR	Internal error
12	SOAP_FAULT	An exception raised by the service
13	SOAP_NO_METHOD	Skeleton error: the skeleton cannot serve the method
14	SOAP_GET_METHOD	Unsupported HTTP GET
15	SOAP_EOM	Out of memory
16	SOAP_NULL	An element was null, while it is not supposed to be null
17	SOAP_MULTI_ID	Multiple occurrences of the same element ID on the input
18	SOAP_MISSING_ID	Element ID missing for an HREF on the input
19	SOAP_HREF	Reference to object is incompatible with the object refered to
20	SOAP_TCP_ERROR	A TCP connection error occured
21	SOAP_HTTP_ERROR	An HTTP error occured
22	SOAP_SSL_ERROR	An SSL error occured
23	SOAP_ZLIB_ERROR	A Zlib error occured
24	SOAP_DIME_ERROR	DIME parsing error
25	SOAP_EOD	End of DIME error
26	SOAP_VERSIONMISMATCH*	SOAP version mismatch or no SOAP message
27	SOAP_DIME_MISMATCH	DIME version mismatch
28	SOAP_PLUGIN_ERROR	Failed to register plugin
29	SOAP_DATAENCODINGUNKNOWN	SOAP 1.2 DataEncodingUnknown fault
-1	SOAP_EOF	Unexpected end of file, no input, or timeout while receiving data

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 soap.error, where soap is a variable that contains the current runtime environment. The function soap_print_fault(**struct** soap *soap, FILE *fd) can be called to display an error message on fd where current value of the soap.error variable is used by the function to display the error. The function soap_print_fault_location(**struct** soap *soap, FILE *fd) prints the location of the error if the error is a result from parsing XML.

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 with the soap_receiver_fault(**struct** soap *soap, **const char** *faultstring, **const char** *detail) which sets the strings soap.fault->faultstring and soap.fault->detail for SOAP 1.1, and soap.fault->SOAP_ENV__Reason and soap.fault->SOAP_ENV__Detail for SOAP 1.2, where soap is a vari-

able that contains the current runtime environment, see Section 10. A receiver error indicates that the service can't handle the request, but can possibly recover from the error. To return an unrecoverable error, use soap_receiver_fault(**struct** soap *soap, **const char** *faultstring, **const char** *detail).

To return a HTTP error code a service method can simply return the HTTP error code number. For example, **return** 404; returns a "404 Not Found" HTTP error back to the client. The soap.error is set to the HTTP error code at the client side. The HTTP 1.1 error codes are:

#	Error
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
307	Temporary Redirect
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Time-out
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Request Entity Too Large
414	Request-URI Too Large
415	Unsupported Media Type
416	Requested range not satisfiable
417	Expectation Failed
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Time-out
505	HTTP Version not supported

The error codes are given for informational purposes only. The HTTP protocol requires the proper actions after an error is issued. gSOAP's HTTP 1.0/1.1 handling is automatic.

8.3 C/C++ Identifier Name to XML Name Translations

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 8.4, see also Section 6.1.2. 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 **struct**s and **class**es with essentially the same XML schema type name, but that have to be distinghuished as seperate C/C++ types.

For decoding, the underscores in identifier names act as wildcards. An XML element is parsed and matches the name of an identifier if the name is identical to the element name (case insensitive) and the underscores in the identifier name are allowed to match any character in the element name. For example, the identifier name I_want__soap_fun_the_bea___DOT_com matches the element name `I-want:SOAP4fun@the-beach.com`.

8.4 Namespace Mapping Table

A namespace mapping table MUST be defined by clients and service applications. The mapping table is used by the serializers and deserializers of the stub and skeleton routines to produce a valid SOAP payload and to validate an incoming SOAP payload. A typical mapping table is shown below:

```
struct Namespace namespaces[] =
{   // {"ns-prefix", "ns-name"}
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"}, // MUST be first
   {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"}, // MUST be second
   {"xsi", "http://www.w3.org/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 a identifier name in the header file specification (see Section 8.3) 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 distinghuished 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:

```
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:

```
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"},
...
```

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

http://tempuri.org in the table. See Section 13.2 on how to specify the namespace URIs in the header file.

For decoding elements with namespace prefixes, the namespace URI associated with the namespace prefix (through the `xmlns` attribute of an XML element) is searched from the beginning to the end in a namespace mapping table, and for every row the following tests are performed as part of the validation process:

1. the string in the second column matches the namespace URI (case insensitive)
2. the string in the optional third column matches the namespace URI (case insensitive), where - is a one-character wildcard and * is a multi-character wildcard

When a match is found, the namespace prefix in the first column of the table is considered semantically identical to the namespace prefix used by the XML element to be decoded, though the prefix names may differ. A service will respond with the namespace that it received from a client in case it matches a pattern in the third column.

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

```
struct a__elt { ... };
struct b__elt { ... };
struct k__elt { ... };
```

and a namespace mapping table in the program:

```
struct Namespace namespaces[] =
{   // {"ns-prefix", "ns-name", "ns-name validation pattern"}
...
   {"a", "some uri"},
   {"b", "other uri"},
   {"c", "his uri", "* uri"},
...
```

Then, the following XML elements will match the structs:

```
<n:elt xmlns:n="some URI">     matches the struct name a__elt
...
<m:elt xmlns:m="other URI">     matches the struct name b__elt
...
<k:elt xmlns:k="my URI">     matches the struct name c__elt
...
```

The response of a service to a client request that uses the namespaces listed above, will include `my URI` for the name space of element `k`.

It is possible to use a number of different namespace tables and select the one that is appropriate. For example, an application might contact many different Web services all using different namespace URIs. If all the URIs are stored in one table, each 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:

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

9 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.

9.1 Primitive Type Encoding

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

Type	XSD Type
bool	`boolean`
char* (C string)	`string`
char	`byte`
double	`double`
float	`float`
int	`int`
long	`long`
LONG64	`long`
long long	`long`
short	`short`
time_t	`dateTime`
unsigned char	`unsignedByte`
unsigned int	`unsignedInt`
unsigned long	`unsignedLong`
ULONG64	`unsignedLong`
unsigned long long	`unsignedLong`
unsigned short	`unsignedShort`
wchar_t*	`string`

Objects of type **void** and **void*** cannot be encoded. Enumerations and bit masks are supported as well, see 9.3.

9.2 How to Encode and Decode Primitive Types as XSD Types

By default, encoding of the primitive types will take place as per SOAP encoding style. The encoding can be changed to any XML schema type (XSD type) with an optional namespace prefix by using a **typedef** in the header file input to the gSOAP stub and skeleton compiler. The declaration

enables the implementation of built-in XML schema types (also known as XSD types) such as `positiveInteger`, `xsd:anyURI`, and `xsd:date` for which no built-in data structures in C and C++ exist but which can be represented using standard data structures such as strings, integers, and floats.

The **typedef** declaration is frequently used for convenience in C. A **typedef** declares a type name for a (complex) type expression. The type name can then be used in other declarations in place of the more complex type expression, which often improves the readability of the program code.

The gSOAP 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 XSD 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 XSD types. To this end, a **typedef** declaration can be used to declare an XSD type.

For example, the declaration

```
typedef unsigned int xsd__positiveInteger;
```

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

```
<positiveInteger xsi:type="xsd:positiveInteger">3</positiveInteger>
```

The built-in primitive and derived numerical XML Schema types are listed below together with their recommended **typedef** declarations. Note that the SOAP encoding schemas for primitive types are derived from the built-in XML schema types, so SOAP_ENC__ can be used as a namespace prefix instead of xsd__.

`xsd:anyURI` Represents a Uniform Resource Identifier Reference (URI). Each URI scheme imposes specialized syntax rules for URIs in that scheme, including restrictions on the syntax of allowed fragement identifiers. It is recommended to use strings to store `xsd:anyURI` XML schema types. The recommended type declaration is:

```
typedef char *xsd__anyURI;
```

`xsd:base64Binary` Represents Base64-encoded arbitrary binary data. For using the `xsd:base64Binary` XML schema type, the use of the base64Binary representation of a dynamic array is **strongly** recommended, see Section 9.10. However, the type can also be declared as a string and the encoding will be string-based:

```
typedef char *xsd__base64Binary;
```

With this approach, it is 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 9.3.5. The corresponding type declaration is:

typedef bool xsd__boolean;

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

```
<xsd:boolean xsi:type="xsd:boolean">...</xsd:boolean>
```

`xsd:byte` Represents a byte (-128...127). The corresponding type declaration is:

typedef char xsd__byte;

Type xsd__byte declares a byte which is encoded as

```
<xsd:byte xsi:type="xsd:byte">...</xsd:byte>
```

`xsd:dateTime` Represents a date and time. The lexical representation is according to the ISO 8601 extended format CCYY-MM-DDThh:mm:ss where "CC" represents the century, "YY" the year, "MM" the month and "DD" the day, preceded by an optional leading "-" sign to indicate a negative number. If the sign is omitted, "+" is assumed. The letter "T" is the date/time separator and "hh", "mm", "ss" represent hour, minute and second respectively. It is recommended to use the time_t type to store `xsd:dateTime` XML schema types and the type declaration is:

typedef time_t xsd__dateTime;

However, note that calendar times before the year 1902 or after the year 2037 cannot be represented. Upon receiving a date outside this range, the time_t value will be set to -1.

Strings (**char***) can be used to store `xsd:dateTime` XML schema types. The type declaration is:

typedef char *xsd__dateTime;

In this case, it is up to the application to read and set the dateTime representation.

`xsd:date` Represents a date. The lexical representation for date is the reduced (right truncated) lexical representation for dateTime: CCYY-MM-DD. It is recommended to use strings (**char***) to store `xsd:date` XML schema types. The type declaration is:

typedef char *xsd__date;

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

typedef double xsd__decimal;

Type xsd__decimal declares a double floating point number which is encoded as

```
<xsd:double xsi:type="xsd:decimal">...</xsd:double>
```

`xsd:double` Corresponds to the IEEE double-precision 64-bit floating point type. The type declaration is:

typedef double xsd__double;

Type xsd__double declares a double floating point number which is encoded as

```
<xsd:double xsi:type="xsd:double">...</xsd:double>
```

`xsd:duration` Represents a duration of time. The lexical representation for duration is the ISO 8601 extended format PnYn MnDTnH nMnS, where nY represents the number of years, nM the number of months, nD the number of days, T is the date/time separator, nH the number of hours, nM the number of minutes and nS the number of seconds. The number of seconds can include decimal digits to arbitrary precision. It is recommended to use strings (**char***) to store `xsd:duration` XML schema types. The type declaration is:

```
typedef char *xsd__duration;
```

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

```
typedef float xsd__float;
```

Type `xsd__float` declares a floating point number which is encoded as

```
<xsd:float xsi:type="xsd:float">...</xsd:float>
```

`xsd:hexBinary` Represents arbitrary hex-encoded binary data. It has a lexical representation where each binary octet is encoded as a character tuple, consisting of two hexadecimal digits ([0-9a-fA-F]) representing the octet code. For example, "0FB7" is a hex encoding for the 16-bit integer 4023 (whose binary representation is 111110110111. For using the `xsd:hexBinary` XML schema type, the use of the hexBinary representation of a dynamic array is **strongly** recommended, see Section 9.11. However, the type can also be declared as a string and the encoding will be string-based:

```
typedef char *xsd__hexBinary;
```

With this approach, it is solely the responsibility of the application to make sure the string content consists of a sequence of octets.

`xsd:int` Corresponds to a 32-bit integer in the range -2147483648 to 2147483647. If the C++ compiler supports 32-bit **int** types, the type declaration can use the **int** type:

```
typedef int xsd__int;
```

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

```
typedef long xsd__int;
```

Type `xsd__int` declares a 32-bit integer which is encoded as

```
<xsd:int xsi:type="xsd:int">...</xsd:int>
```

`xsd:integer` Corresponds to an unbounded integer. Since C++ does not support unbounded integers as a standard feature, the recommended type declaration is:

```
typedef long long xsd__integer;
```

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

```
<xsd:integer xsi:type="xsd:integer">...</xsd:integer>
```

Another possibility is to use strings to represent unbounded integers and do the translation in code.

`xsd:long` Corresponds to a 64-bit integer in the range -9223372036854775808 to 9223372036854775807. The type declaration is:

```
typedef long long xsd__long;
```

Or in Visual C++:

```
typedef LONG64 xsd__long;
```

Type xsd__long declares a 64-bit integer which is encoded as

```
<xsd:long xsi:type="xsd:long">...</xsd:long>
```

`xsd:negativeInteger` Corresponds to a negative unbounded integer (< 0). Since C++ does not support unbounded integers as a standard feature, the recommended type declaration is:

```
typedef long long xsd__negativeInteger;
```

Type xsd__negativeInteger declares a 64-bit integer which is encoded as a `xsd:negativeInteger`:

```
<xsd:negativeInteger xsi:type="xsd:negativeInteger">...</xsd:negativeInteger>
```

Another possibility is to use strings to represent unbounded integers and do the translation in code.

`xsd:nonNegativeInteger` Corresponds to a non-negative unbounded integer (> 0). Since C++ does not support unbounded integers as a standard feature, the recommended type declaration is:

```
typedef unsigned long long xsd__nonNegativeInteger;
```

Type xsd__nonNegativeInteger declares a 64-bit unsigned integer which is encoded as a non-negative unbounded `xsd:nonNegativeInteger`:

```
<xsd:nonNegativeInteger xsi:type="xsd:nonNegativeInteger">...</xsd:nonNegativeInteger>
```

Another possibility is to use strings to represent unbounded integers and do the translation in code.

`xsd:nonPositiveInteger` Corresponds to a non-positive unbounded integer (≤ 0). Since C++ does not support unbounded integers as a standard feature, the recommended type declaration is:

```
typedef long long xsd__nonPositiveInteger;
```

Type xsd__nonPositiveInteger declares a 64-bit integer which is encoded as a `xsd:nonPositiveInteger`:

```
<xsd:nonPositiveInteger xsi:type="xsd:nonPositiveInteger">...</xsd:nonPositiveInteger>
```

Another possibility is to use strings to represent unbounded integers and do the translation in code.

`xsd:normalizedString` Represents normalized character strings. Normalized character strings do not contain the carriage return (#xD), line feed (#xA) nor tab (#x9) characters. It is recommended to use strings to store `xsd:normalizeString` 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:

```
typedef char *xsd__token;
```

Type xsd__token declares a string type which is encoded as

```
<xsd:token xsi:type="xsd:token">...</xsd:token>
```

It is solely the responsibility of the application to make sure the strings do not contain the line feed (#xA) nor tab (#x9) characters, that have no leading or trailing spaces (#x20) and that have no internal sequences of two or more spaces.

`xsd:unsignedByte` Corresponds to an 8-bit unsigned integer in the range 0 to 255. The type declaration is:

```
typedef unsigned char xsd__unsignedByte;
```

Type xsd__unsignedByte declares a unsigned 8-bit integer which is encoded as

```
<xsd:unsignedByte xsi:type="xsd:unsignedByte">...</xsd:unsignedByte>
```

`xsd:unsignedInt` Corresponds to a 32-bit unsigned integer in the range 0 to 4294967295. If the C++ compiler supports 32-bit **int** types, the type declaration can use the **int** type:

```
typedef unsigned int xsd__unsignedInt;
```

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

```
typedef unsigned long xsd__unsignedInt;
```

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

```
<xsd:unsignedInt xsi:type="xsd:unsignedInt">...</xsd:unsignedInt>
```

`xsd:unsignedLong` Corresponds to a 64-bit unsigned integer in the range 0 to 18446744073709551615. The type declaration is:

```
typedef unsigned long long xsd__unsignedLong;
```

Or in Visual C++:

```
typedef ULONG64 xsd__unsignedLong;
```

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

```
<xsd:unsignedLong xsi:type="xsd:unsignedLong">...</xsd:unsignedLong>
```

`xsd:unsignedShort` Corresponds to a 16-bit unsigned integer in the range 0 to 65535. The type declaration is:

```
typedef unsigned short xsd__unsignedShort;
```

Type xsd__unsginedShort declares an unsigned short 16-bit integer which is encoded as

```
<xsd:unsignedShort xsi:type="xsd:unsignedShort">...</xsd:unsignedShort>
```

Other XML schema types such as `gYearMonth`, `gYear`, `gMonthDay`, `gDay`, `xsd:gMonth`, `QName`, `NOTATION`, etc., can be encoded similarly using a **typedef** declaration.

9.2.1 How to Use Multiple C/C++ Types for a Single Primitive XSD Type

Trailing underscores (see Section 8.3) can be used in the type name in a **typedef** to enable the declaration of multiple storage formats for a single XML schema type. For example, one part of a C/C++ application's data structure may use plain strings while another part may use wide character strings. To enable this simultaneous use, declare:

```
typedef char *xsd__string;
typedef wchar_t *xsd__string_;
```

Now, the xsd__string and xsd__string_ types will both be encoded and decoded as XML string types and the use of trailing underscores allows multiple declarations for a single XML schema type.

9.2.2 How to use Wrapper Classes 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` (`xsd:ur-type` in the older 1999 schema). So, for example, an array of `xsd:anyType` in SOAP may actually contain any mix of element types that are the derived types of the root type. The use of polymorphic types is indicated by the WSDL and schema descriptions of a Web service and can therefore be predicted/expected for each particular case.

On the one hand, the **typedef** construct provides a convenient way to associate C/C++ types with XML schema types and makes it easy to incorporate these types in a (legacy) C/C++ application. However, on the other hand the **typedef** declarations cannot be used to support polymorphic XML schema types. Most SOAP clients and services do not use polymorphic types. In case they do, the primitive polymorphic types can be declared as a hierarchy of C++ **class**es that can be used simultaneously with the **typedef** declarations.

The general form of a primitive type declaration that is derived from a super type is:

```
class xsd__type_name: [public xsd__super_type_name]
{ public: Type __item;
  [public:] [private] [protected:]
  method1;
  method2;
  ...
};
```

where Type is a primitive C type. The __item field MUST be the first field in this wrapper class.

For example, the XML schema type hierarchy can be copied to C++ with the following declarations:

```
class xsd__anyType { };
class xsd__anySimpleType: public xsd__anyType { };
typedef char *xsd__anyURI;
```

```
class xsd__anyURI_: public xsd__anySimpleType { public: xsd__anyURI __item; };
typedef bool xsd__boolean;
class xsd__boolean_: public xsd__anySimpleType { public: xsd__boolean __item; };
typedef char *xsd__date;
class xsd__date_: public xsd__anySimpleType { public: xsd__date __item; };
typedef time_t xsd__dateTime;
class xsd__dateTime_: public xsd__anySimpleType { public: xsd__dateTime __item; };
typedef double xsd__double;
class xsd__double_: public xsd__anySimpleType { public: xsd__double __item; };
typedef char *xsd__duration;
class xsd__duration_: public xsd__anySimpleType { public: xsd__duration __item; };
typedef float xsd__float;
class xsd__float_: public xsd__anySimpleType { public: xsd__float __item; };
typedef char *xsd__time;
class xsd__time_: public xsd__anySimpleType { public: xsd__time __item; };
typedef char *xsd__decimal;
class xsd__decimal_: public xsd__anySimpleType { public: xsd__decimal __item; };
typedef char *xsd__integer;
class xsd__integer_: public xsd__decimal_ { public: xsd__integer __item; };
typedef LONG64 xsd__long;
class xsd__long_: public xsd__integer_ { public: xsd__long __item; };
typedef long xsd__int;
class xsd__int_: public xsd__long_ { public: xsd__int __item; };
typedef short xsd__short;
class xsd__short_: public xsd__int_ { public: xsd__short __item; };
typedef char xsd__byte;
class xsd__byte_: public xsd__short_ { public: xsd__byte __item; };
typedef char *xsd__nonPositiveInteger;
class xsd__nonPositiveInteger_: public xsd__integer_ { public: xsd__nonPositiveInteger __item; };
typedef char *xsd__negativeInteger;
class xsd__negativeInteger_: public xsd__nonPositiveInteger_ { public: xsd__negativeInteger __item;
};
typedef char *xsd__nonNegativeInteger;
class xsd__nonNegativeInteger_: public xsd__integer_ { public: xsd__nonNegativeInteger __item; };
typedef char *xsd__positiveInteger;
class xsd__positiveInteger_: public xsd__nonNegativeInteger_ { public: xsd__positiveInteger __item;
};
typedef ULONG64 xsd__unsignedLong;
class xsd__unsignedLong_: public xsd__nonNegativeInteger_ { public: xsd__unsignedLong __item;
};
typedef unsigned long xsd__unsignedInt;
class xsd__unsignedInt_: public xsd__unsginedLong_ { public: xsd__unsignedInt __item; };
typedef unsigned short xsd__unsignedShort;
class xsd__unsignedShort_: public xsd__unsignedInt_ { public: xsd__unsignedShort __item; };
typedef unsigned char xsd__unsignedByte;
class xsd__unsignedByte_: public xsd__unsignedShort_ { public: xsd__unsignedByte __item; };
typedef char *xsd__string;
class xsd__string_: public xsd__anySimpleType { public: xsd__string __item; };
typedef char *xsd__normalizedString;
class xsd__normalizedString_: public xsd__string_ { public: xsd__normalizedString __item; };
typedef char *xsd__token;
class xsd__token_: public xsd__normalizedString_ { public: xsd__token __item; };
```

Note the use of the trailing underscores for the **class** names to distinhuish 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:base64Binray 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 9.10 and 9.11.

Methods are allowed to be added to the classes above, such as constructors and getter/setter methods, see Section 9.5.4.

Wrapper structs are supported as well, similar to wrapper classes. But thay cannot be used to implement polymorphism. Rather, the wrapper structs facilitate the use of XML attributes with a primitive typed object, see 9.5.7.

9.2.3 XML Schema Type Decoding Rules

The decoding rules for the primitive C and C++ data types is given in the table below:

Type	Allows Decoding of	Precision Lost?
bool	[xsd:]boolean	no
char* (C string)	any type, see 9.2.5	no
wchar_t * (wide string)	any type, see 9.2.5	no
double	[xsd:]double	no
	[xsd:]float	no
	[xsd:]long	no
	[xsd:]int	no
	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedLong	no
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
	[xsd:]decimal	possibly
	[xsd:]integer	possibly
	[xsd:]positiveInteger	possibly
	[xsd:]negativeInteger	possibly
	[xsd:]nonPositiveInteger	possibly
	[xsd:]nonNegativeInteger	possibly
float	[xsd:]float	no
	[xsd:]long	no
	[xsd:]int	no
	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedLong	no
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
	[xsd:]decimal	possibly
	[xsd:]integer	possibly
	[xsd:]positiveInteger	possibly
	[xsd:]negativeInteger	possibly
	[xsd:]nonPositiveInteger	possibly
	[xsd:]nonNegativeInteger	possibly
long long	[xsd:]long	no
	[xsd:]int	no
	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedLong	possibly
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
	[xsd:]integer	possibly
	[xsd:]positiveInteger	possibly
	[xsd:]negativeInteger	possibly
	[xsd:]nonPositiveInteger	possibly
	[xsd:]nonNegativeInteger	possibly

Type	Allows Decoding of	Precision Lost?
long	[xsd:]long	possibly, if **long** is 32 bit
	[xsd:]int	no
	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedLong	possibly
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
int	[xsd:]int	no
	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedInt	possibly
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
short	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
char	[xsd:]byte	no
	[xsd:]unsignedByte	possibly
unsigned long long	[xsd:]unsignedLong	no
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
	[xsd:]positiveInteger	possibly
	[xsd:]nonNegativeInteger	possibly
unsigned long	[xsd:]unsignedLong	possibly, if **long** is 32 bit
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
unsigned int	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
unsigned short	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
unsigned char	[xsd:]unsignedByte	no
time_t	[xsd:]dateTime	no(?)

Due to limitations in representation of certain primitive C++ types, a possible loss of accuracy may occur with the decoding of certain 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:

```
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 a **unsigned long long** data type.

9.2.4 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

```
char *s = "hello", *t = s;
```

The `s` and `t` variables are assigned the same string, and when serialized, `t` refers to the content of `s`:

```
<string id="123" xsi:type="string">hello</string>
...
<string href="#123"/>
```

The example assumed that `s` and `t` are encoded as independent elements.

Note: the use of **typedef** to declare a string type such as xsd__string will not affect the multi-reference string encoding. However, strings declared with different **typedef**s will never be considered multi-reference even when they point to the same string. For example

```
typedef char *xsd__string;
typedef char *xsd__anyURI;
xsd__anyURI *s = "http://www.myservice.com";
xsd__string *t = s;
```

The variables `s` and `t` point to the same string, but since they are considered different types their content will not be shared in the SOAP payload through a multi-referenced string.

9.2.5 "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:

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

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

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/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></picture>`".

9.2.6 STL Strings

gSOAP supports STL strings std::string and std::wstring. For example:

```
typedef std::string xsd__string;
class ns__myClass
{ public:
  xsd__string s; // serialized with xsi:type="xsd:string"
  std::string t; // serialized without xsi:type
...
};
```

Caution: Please avoid mixing std::string and C strings (**char***) in the header file when using SOAP 1.1 encoding. The problem is that multi-referenced strings in SOAP encoded messages cannot be assigned simultaneously to a std::string and a **char*** string.

9.2.7 Changing the Encoding Precision of float and double Types

The double encoding format is by default set to "%.18G" (see a manual on printf text formatting in C), i.e. at most 18 digits of precision to limit a loss in accuracy. The float encoding format is by default "%.9G", i.e. at most 9 digits of precision.

The encoding format of a double type can be set by assigning a format string to soap.double_format, where soap is a variable that contains the current runtime environment. For example:

```
struct soap soap;
soap_init(&soap); // sets double_format = "%.18G"
soap.double_format = "%e"; // redefine
```

which causes all doubles to be encoded in scientific notation. Likewise, the encoding format of a float type can be set by assigning a format string to the static soap_float_format string variable. For example:

```
struct soap soap;
soap_init(&soap); // sets float_format = "%.9G"
soap.float_format = "%.4f"; // redefine
```

which causes all floats to be encoded with four digits precision.

Caution: The format strings are not automatically reset before or after SOAP communications. An error in the format string may result in the incorrect encoding of floating point values.

9.2.8 INF, -INF, and NaN Values of float and double Types

The gSOAP runtime stdsoap2.cpp and header file stdsoap2.h support the marshalling of IEEE INF, -INF, and NaN representations. Under certain circumstances this may break if the hardware and/or C/C++ compiler does not support these representations. To remove the representations, remove the inclusion of the ¡math.h¿ header file from the stdsoap2.h file. You can control the representations as well, which are defined by the macros:

```
#define FLT_NAN
#define FLT_PINFTY
#define FLT_NINFTY
#define DBL_NAN
#define DBL_PINFTY
#define DBL_NINFTY
```

9.3 Enumeration Type Encoding and Decoding

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

9.3.1 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:

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

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

```
<weekday xsi:type="weekday">Mon</weekday>
```

The value of the `xsi:type` attribute is the enumeration-type identifier's name. If the element is independent as in the example above, the element name is the enumeration-type identifier's name.

The encoding of complex types such as enumerations requires a reference to an XML schema through the use of a namespace prefix. The namespace prefix can be specified as part of the

enumeration-type identifier's name, with the usual namespace prefix conventions for identifiers. This can be used to explicitly specify the encoding style. For example:

```
enum ns1__weekday {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

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

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

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

```
<xsd:element name="weekday" type="tns:weekday"/>
<xsd:simpleType name="weekday">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Mon"/>
    <xsd:enumeration value="Tue"/>
    <xsd:enumeration value="Wed"/>
    <xsd:enumeration value="Thu"/>
    <xsd:enumeration value="Fri"/>
    <xsd:enumeration value="Sat"/>
    <xsd:enumeration value="Sun"/>
  </xsd:restriction>
</xsd:simpleType>
```

9.3.2 Encoding of Enumeration Constants

If the value of an enumeration-typed variable has no corresponding named constant, the value is encoded as a signed integer literal. For example, the following declaration of a workday enumeration type lacks named constants for Saturday and Sunday:

```
enum ns1__workday {Mon, Tue, Wed, Thu, Fri};
```

If the constant 5 (Saturday) or 6 (Sunday) is assigned to a variable of the workday enumeration type, the variable will be encoded with the integer literals 5 and 6, respectively. For example:

```
<ns1:workday xsi:type="ns1:workday">5</ns1:workday>
```

Since this is legal in C++ and SOAP allows enumeration constants to be integer literals, this method ensures that non-symbolic enumeration constants are correctly communicated to another party if the other party accepts literal enumeration constants (as with the gSOAP 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:

```
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.

9.3.3 Initialized Enumeration Constants

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

```
enum ns1__relation {LESS = -1, EQUAL = 0, GREATER = 1};
```

The symbolic names `LESS`, `EQUAL`, and `GREATER` will appear in the SOAP payload for the encoding of the `ns1__relation` enumeration values -1, 0, and 1, respectively.

9.3.4 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 distinghuish it from other symbolic names in C++. This guarantees that the SOAP encoding will use the same name, while the symbolic names can be distinghuished in C++. Effectively, the underscores are removed from a symbolic name prior to encoding.

Consider for example:

```
enum ns1__workday {Mon, Tue, Wed, Thu, Fri};
enum ns1__weekday {Mon_, Tue_, Wed_, Thu_, Fri_, Sat_, Sun_};
```

which will result in the encoding of the constants of `enum ns1__weekday` without the underscore, for example as `Mon`.

Caution: The following declaration:

```
enum ns1__workday {Mon, Tue, Wed, Thu, Fri};
enum ns1__weekday {Sat = 5, Sun = 6};
```

will not properly encode the `weekday` enumeration, because it lacks the named constants for `workday` in its enumeration list.

9.3.5 Boolean Enumeration Type Encoding and Decoding for C Compilers

When a pure C compiler is used to create SOAP clients and services, the `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 `boolean` XML schema type supports the mathematical concept of binary-valued logic. The `boolean` XML schema encoding style can be specified by using the `xsd` prefix. For example:

```
enum xsd__boolean {false_, true_};
```

The value `false_`, for example, is encoded as:

```
<xsd:boolean xsi:type="xsd:boolean">false</xsd:boolean>
```

Peculiar of the SOAP boolean type encoding is that it only defines the values `0` and `1`, while the built-in XML schema boolean type also defines the `false` and `true` symbolic constants as valid values. The following example declaration of an enumeration type lacks named constants altogether to force encoding of the enumeration values as literal constants:

```
enum SOAP_ENC__boolean {};
```

The value `0`, for example, is encoded with an integer literal:

```
<SOAP-ENC:boolean xsi:type="SOAP-ENC:boolean">0<SOAP-ENC:boolean>
```

9.3.6 Bitmask Enumeration Encoding and Decoding

A bitmask is an enumeration of flags such as declared with C#'s [Flags] **enum** annotation. gSOAP supports bitmask encoding and decoding for interoperability. However, bitmask types are not standardized with SOAP RPC.

A special syntactic convention is used in the header file input to the gSOAP compiler to indicate the use of bitmasks with an asterisk:

enum * *name* { *enum-constant*, *enum-constant*, ... };

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:

```
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:

```
enum ns__machineStatus { ON=1, BELT=2, VALVE=4, HATCH=8};
```

A remote method implementation in a Web service can return:

```
int ns__getMachineStatus(struct soap *soap, char *name, enum ns__machineStatus result)
{ ...
  *result = BELT — HATCH;
  return SOAP_OK;
}
```

9.4 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 9.5 for further details. However, the encoding and decoding of **struct**s is more efficient compared to **class** instances due to the lack of inheritance and the requirement by the marshalling routines to check inheritance properties at run time.

Certain type of fields of a **struct** can be (de)serialized as XML attributes. See 9.5.7 for more details.

9.5 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:

```
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 8.3)

class_name1 is the element name of the compound data type (see identifier translation rules 8.3).

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 ommitted. 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 undistinghuishable (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.

Classes should be declared "volatile" if you don't want gSOAP to add serialization methods to these classes, see Section 13.6 for more details.

Class templates are not supported by the gSOAP compiler, but you can use STL containers, see Section 9.9.7. You can also define your own containers similar to STL containers.

Certain fields of a **class** can be (de)serialized as XML attributes. See 9.5.7 for more details.

Arrays may be embedded within a class (and struct) using a pointer field and size information, see Section 9.9.6. This defines what is sometimes referred to in SOAP as "generics".

Void pointers may be used in a class (or struct), but you have to add a type field so the gSOAP runtime can determine the type of object pointed to, see Section 9.7.

A **class** instance is encoded as:

```
<[namespace-prefix:]class-name xsi:type="[namespace-prefix:]class-name">
<basefield-name1 xsi:type="...">...</basefield-name1>
<basefield-name2 xsi:type="...">...</basefield-name2>
...
<field-name1 xsi:type="...">...</field-name1>
<field-name2 xsi:type="...">...</field-name2>
...
</[namespace-prefix:]class-name>
```

where the `field-name` accessors have element-name representations of the class fields and the `basefield-name` accessors have element-name representations of the base class fields. (The optional parts resulting from the specification are shown enclosed in [].)

The decoding of a class instance allows any ordering of the accessors in the SOAP payload. However, if a base class field name is identical to a derived class field name because the field is overloaded, the base class field name MUST precede the derived class field name in the SOAP payload for decoding. gSOAP guarantees this, but interoperability with other SOAP implementations is cannot be guaranteed.

9.5.1 Example

The following example declares a base class ns__Object and a derived class ns__Shape:

```
// Contents of file "shape.h":
class ns__Object
{
  public:
  char *name;
};
class ns__Shape : public ns__Object
{
  public:
```

```
  int sides;
  enum ns__Color {Red, Green, Blue} color;
  ns__Shape();
  ns__Shape(int sides, enum ns__Green color);
  ~ns__Shape();
};
```

The implementation of the methods of **class** ns__Shape must not be part of the header file and need to be defined elsewhere.

An instance of **class** ns__Shape with name Triangle, 3 sides, and color Green is encoded as:

```
<ns:Shape xsi:type="ns:Shape">
<name xsi:type="string">Triangle</name>
<sides xsi:type="int">3</sides>
<color xsi:type="ns:Color">Green</color>
</ns:shape>
```

The namespace URI of the namespace prefix `ns` must be defined by a namespace mapping table, see Section 8.4.

9.5.2 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:

```
// Contents of file "triangle.h":
class ns__Triangle : public ns__Object
{
  public:
  int size;
  static const int sides = 3;
};
```

An instance of **class** ns__Triangle is encoded in SOAP as:

```
<ns:Triangle xsi:type="ns:Triangle">
<name xsi:type="string">Triangle</name>
<size xsi:type="int">15</size>
<sides xsi:type="int">3>/sides>
</ns:Triangle>
```

Decoding will ignore the sides field's value.

Caution: The current gSOAP implementation does not support encoding **static const** fields, due to C++ compiler compatibility differences. This feature may be provided the future.

9.5.3 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**.

9.5.4 Getter and Setter Methods

Setter and getter methods are invoked at run time upon serialization and deserialization of class instances, recpectively. The use of setter and getter methods adds more flexibility in to the serialization and deserialization process.

A setter method is called in the serialization phase from the virtual soap_serialization method generated by the gSOAP compiler. You can use setter methods to process a class instance just before it is serialized. A setter method can be used to convert application data, such as translating transient application data into serializable data, for example. You can also use setter methods to retrieve dynamic content and use it to update a class instance right before serializaton. It helps to remember setters as "set to serialize" operations.

Getter methods are invoked after deserialization of the instance. You can use them to adjust the contents of class instances after all their members have been deserialized. Getters can be used to convert deserialized members into transient members and even invoke methods to process the deserialized data on the fly.

Getter and setter methods have the following signature:

```
[virtual] int get(struct soap *soap) [const];
[virtual] int set(struct soap *soap);
```

The active soap struct will be passed to the get and set methods. The methods should return SOAP_OK when successful. A setter method should prepare the contents of the class instance for serialization. A getter method should process the instance after deserialization.

Here is an example of a base64 binary class:

```
class xsd__base64Binary
{ public:
  unsigned char *__ptr;
  int __size;
  int get(struct soap *soap);
  int set(struct soap *soap);
};
```

Suppose that the type and options members of the attachment should be set when the class is about to be serialized. This can be accomplished with the set method from the information provided by the __ptr to the data and the soap struct passed to the set method (you can pass data via the **void***soap.user field).

The get method is invoked after the base64 data has been processed. You can use it for post-processing purposes.

Here is another example. It defines a primitive update type. The class is a wrapper for the time_t type, see Section 9.2.2. Therefore, elements of this type contain `xsd:dateType` data.

```
class update
{ public:
  time_t __item;
  int set(struct soap *soap);
};
```

The setter method assigns the current time:

```
int update::set(struct soap *soap)
{
  this->__item = time(NULL);
  return SOAP_OK;
}
```

Therefore, serialization results in the inclusion of a time stamp in XML.

Caution: a get method is invoked only when the XML element with its data was completely parsed. The method is not invoked when the element is an `xsi:nil` element or has an `href` attribute.

Caution: The soap_serialize method of a class calls the setter (when provided). However, the soap_serialize method is declared **const** while the setter should be allowed to modify the contents of the class instance. Therefore, the gSOAP-generated code recasts the instance and the **const** is removed when invoking the setter.

9.5.5 Streaming XML with Getter and Setter Methods

Getter methods enable streaming XML operations. A getter method is invoked when the object is deserialized and the rest of the SOAP/XML message has not been processed yet. For example, you can add a getter method to the SOAP Header class to implement header processing logic that is activated as soon as the SOAP Header is received. An example code is shown below:

```
class h__Authentication
{ public:
  char *id;
  int get(struct soap *soap);
};
class SOAP_ENV__Header
{ public:
  h__Authentication *h__authentication;
};
```

The Authentication SOAP Header field is instantiated and decoded. After decoding, the getter method is invoked, which can be used to check the id before the rest of the SOAP message is processed.

9.5.6 Polymorphism, Derived Classes, and Dynamic Binding

Interoperability between client and service applications developed with gSOAP is established even when clients and/or services use derived classes instead of the base classes used in the declaration of the 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 classes 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:

```
// 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:

```
// 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:

```
// CLIENT
#include "soapH.h"
int main()
{
  struct soap soap;
  soap_init(&soap);
  Derived obj1;
  Base *obj2;
  struct methodResponse r;
  obj1.name = "X";
  obj1.num = 3;
  soap_call_method(&soap, url, action, &obj1, r);
  r.obj2->print();
}
...
```

The following example SERVER1 application copies a class instance (Base or Derived class) from the input to the output parameter:

```
// SERVER1
#include "soapH.h"
int main()
{
  soap_serve(soap_new());
}
int method(struct soap *soap, Base *obj1, struct methodResponse &result)
{
  obj1->print();
  result.obj2 = obj1;
  return SOAP_OK;
}
...
```

The following messages are produced by the CLIENT and SERVER1 applications:

```
CLIENT: created a Derived class instance
SERVER1: created a Derived class instance
SERVER1: print(): Derived class instance X 3
CLIENT: created a Derived class instance
CLIENT: print(): Derived class instance X 3
```

Which indicates that the derived class kept its identity when it passed through SERVER1. Note that instances are created both by the CLIENT and SERVER1 by the demarshalling process.

Now suppose a service application is developed that only accepts Base class instances. The header file is:

```
// Contents of file "base.h":
class Base
{
  public:
  char *name;
  Base();
  virtual void print();
};
int method(Base *in, Base *out);
```

This header file specification is processed by the gSOAP 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:

```
// Method implementations of the Base class:
#include "soapH.h"
...
Base::Base()
{
  cout << "created a Base class instance" << endl;
}
Base::print()
{
  cout << "print(): Base class instance " << name << endl;
}
```

And the SERVER2 application is that uses the Base class is:

```
// SERVER2
#include "soapH.h"
int main()
{
  soap_serve(soap_new());
}
int method(struct soap *soap, Base *obj1, struct methodResponse &result)
{
  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.

9.5.7 XML Attributes

The SOAP RPC/LIT and SOAP DOC/LIT encoding styles support XML attributes in SOAP messages while SOAP RPC with "Section 5" encoding does not support XML attributes other than the SOAP and XSD specific attributes. SOAP RPC "Section 5" encoding has advantages for cross-language interoperability and data encodings such as graph serialization. However, RPC/LIT and DOC/LIT enables direct exchange of XML documents, which may include encoded application data structures. Language interoperability is compromized, because no mapping between XML and the typical language data types is defined. The meaning of the RPC/LIT and DOC/LIT XML content is Schema driven rather than application/language driven.

gSOAP supports XML attribute (de)serialization of members in structs and classes. Attributes are primitive XSD types, such as strings, enumerations, boolean, and numeric types. To declare an XML attribute in a struct/class, the qualifier `@` is used with the type of the attribute. The type must be primitive type (including enumerations and strings), which can be declared with or without a **typedef** to associate a XSD type with the C/C+ type. For example

```
typedef char *xsd__string;
typedef bool *xsd__boolean;
enum ns__state { _0, _1, _2 };
struct ns__myStruct
{
  @ xsd__string ns__type; // encode as XML attribute 'ns:type' of type 'xsd:string'
  @ xsd__boolean ns__flag = false; // encode as XML attribute 'ns:flag' of type 'xsd:boolean'
  @ enum ns__state ns__state = _2; // encode as XML attribute 'ns:state' of type 'ns:state'
  struct ns__myStruct *next;
};
```

The `@` qualifier indicates XML attribute encoding for the `ns__type`, `ns__flag`, and `ns__state` fields. Note that the namespace prefix `ns` is used to distinghuish these attributes from any other attributes such as `xsi:type` (`ns:type` is not to be confused with `xsi:type`).

Default values can be associated with any field that has a primitive type in a struct/class, as is illustrated in this example. The default values are used when the receiving message does not contain the corresponding values.

String attributes are optional. Other type of attributes should be declared as pointers to make them optional:

```
struct ns__myStruct
{
  @int *a; // omitted when NULL };
```

Because a remote method request and response is essentially a struct, XML attributes can also be associated with method requests and responses. For example

```
int ns__myMethod(@char *ns__name, ...);
```

Attributes can also be attached to the dynamic arrays, binary types, and wrapper classes/structs of primitive types. Wrapper classes are described in Section 9.2.2. For example

```
struct xsd__string
{
  char *__item;
  @ xsd__boolean flag;
};
```

and

```
struct xsd__base64Binary
{
  unsigned char *__ptr;
  int __size;
  @ xsd__boolean flag;
};
```

The attribute declarations MUST follow the `__item`, `__ptr`, and `__size` fields which define the characteristics of wrapper structs/classes and dynamic arrays.

Caution: Do not use XML attributes with SOAP RPC encoding. You can only use attributes with RPC literal encoding.

9.5.8 QName Attributes and Elements

gSOAP ensures the proper decoding of XSD QNames. An element or attribute with type QName (Qualified Name) contains a namespace prefix and a local name. You can declare a QName type as a **typedef char** *xsd__QName. Values of type QName are internally handled as regular strings. gSOAP takes care of the proper namespace prefix mappings when deserializing QName values. For example

```
typedef char *xsd__QName;
struct ns__myStruct
{
  xsd__QName elt = "ns:xyz"; // QName element with default value "ns:xyz"
  @ xsd__QName att = "ns:abc"; // QName attribute with default value "ns:abc"
};
```

When the elt and att fields are serialized, their string contents are just transmitted (which means that the application is responsable to ensure proper formatting of the QName strings prior to transmission). When the fields are deserialized however, gSOAP takes care mapping the qualifiers to the appropriate namespace prefixes. Suppose that the inbound value for the elt is `x:def`, where the namespace name associated with the prefix `x` matches the namespace name of the prefix `ns` (as defined in the namespace mapping table). Then, the value is automatically converted into `ns:def`. If the namespace name is not in the table, then `x:def` is converted to "URI":def where `"URI"` is the namespace URI bound to `x` in the message received. This enables an application to retrieve the namespace information, whether it is in the namespace mapping table or not.

Note: QName is a pre-defined typedef type and used by gSOAP to (de)serialize SOAP Fault codes which are QName elements.

9.6 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 indistinghuishable from the encoded representation of the data type pointed to.

9.6.1 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 7.12 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:

```
typedef char *xsd__string;
struct ns__list
{
  xsd__string value;
  struct ns__list *next;
};
```

Suppose a cyclic linked list is created. The first node contains the value "`abc`" and points to a node with value "`def`" which in turn points to the first node. This is encoded as:

```
<ns:list id="1" xsi:type="ns:list">
  <value xsi:type="xsd:string">abc</value>
  <next xsi:type="ns:list">
    <value xsi:type="xsd:string">def</value>
    <next href="#1"/>
  </next>
</ns:list>
```

In case multi-referenced data is received that "does not fit in a pointer-based structure", the data is copied. For example, the following two **struct**s are similar, except that the first uses pointer-based fields while the other uses non-pointer-based fields:

```
typedef long xsd__int;
struct ns__record
{
  xsd__int *a;
  xsd__int *b;
} P;
struct ns__record
{
  xsd__int a;
  xsd__int b;
} R;
...
  P.a = &n;
  P.b = &n;
...
```

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

```
<ns:record xsi:type="ns:record">
  <a href="#1"/>
  <b href="#1"/>
</ns:record>
<id id="1" xsi:type="xsd:int">123</id>
```

Now, the decoding of the content in the R data structure that does not use pointers to integers results in a copy of each multi-reference integer. Note that the two **struct**s resemble the same XML data type because the trailing underscore will be ignored in XML encoding and decoding.

9.6.2 NULL Pointers and Nil Elements

A NULL pointer is **not** serialized, unless the pointer itself is pointed to by another pointer (but see Section 7.12 to control the serialization of NULLs). For example:

```
struct X
{
  int *p;
  int **q;
}
```

Suppose pointer q points to pointer p and suppose p=NULL. In that case the p pointer is serialized as

```
<...  id="123" xsi:nil="true"/>
```

and the serialization of q refers to `href="#123"`. Note that SOAP 1.1 does not support pointer to pointer types (!), so this encoding is specific to gSOAP. The pointer to pointer encoding is rarely used in codes anyway. More common is a pointer to a data type such as a **struct** with pointer fields.

Caution: When the deserializer encounters an XML element that has a `xsi:nil="true"` attribute but the corresponding C++ data is not a pointer or reference, the deserializer will terminate with a SOAP_NULL fault when the SOAP_XML_NIL flag is set. The types section of a WSDL description contains information on the "nilability" of data.

9.7 Void Pointers

In general, void pointers (**void***) cannot be (de)serialized because the data they are pointing to is untyped. To enable the (de)serialization of the void pointers that are members of structs or classes, you have to insert a __type field right before the void pointer field. The **int** __type field contains run time information on the type of the data pointed to by **void*** member in a struct/class to enable the (de)serialization of this data. The **int** __type field is set to a SOAP_TYPE_X value, where X is the name of a type. gSOAP generates the SOAP_TYPE_X definitions in soapH.h and uses them internally to uniquely identify the type of each object. The type naming conventions outlined in Section 6.4.1 are used to determine the type name for *X*.

Here is an example to illustrate the (de)serialization of a **void*** field in a struct:

```
struct myStruct
{
  int __type; // the SOAP_TYPE pointed to by p
  void *p;
};
```

The `__type` integer can be set to 0 at run time to omit the serialization of the void pointer field.

The following example illustrates the initialization of myStruct with a void pointer to an int:

```
struct myStruct S;
int n;
S.p = &n;
S.__type = SOAP_TYPE_int;
```

The serialized output of S contains the integer.

The deserializer for myStruct will automatically set the `__type` field and void pointer to the deserialized data, provided that the XML content for `p` carries the `xsi:type` attribute from which gSOAP can determine the type.

Important: when (de)serializing strings via a **void*** field, the **void*** pointer MUST directly point to the string value rather than indirectly as with all other types. For example:

```
struct myStruct S;
S.p = (void*)"Hello";
S.__type = SOAP_TYPE_string;
```

This is the case for all string-based types, including types defined with **typedef char***.

You may use an arbitrary suffix with the `__type` fields to handle multiple void pointers in structs/classes. For example

```
struct myStruct
{
  int __typeOfp; // the SOAP_TYPE pointed to by p
  void *p;
  int __typeOfq; // the SOAP_TYPE pointed to by q
  void *q;
};
```

Because service method parameters are stored within structs, you can use `__type` and **void*** parameters to pass polymorphic arguments without having to define a C++ class hierarchy (Section 9.5.6). For example:

```
typedef char *xsd__string;
typedef int xsd__int;
typedef float xsd__float;
enum ns__status { on, off };
struct ns__widget { xsd__string name; xsd__int part; }; int ns__myMethod(int __type, void *data,
struct ns__myMethodResponse { int __type; void *return_; } *out);
```

This method has a polymorphic input parameter data and a polymorphic output parameter return_. The __type parameters can be one of SOAP_TYPE_xsd__string, SOAP_TYPE_xsd__int, SOAP_TYPE_xsd__float, SOAP_TYPE_ns__status, or SOAP_TYPE_ns__widget. The WSDL produced by the gSOAP compiler declares the polymorphic parameters of type `xsd:anyType` which is "too loose" and doesn't allow the gSOAP importer to handle the WSDL accurately. Future gSOAP releases might replace `xsd:anyType` with a `choice` schema type that limites the choice of types to the types declared in the header file.

9.8 Fixed-Size Arrays

Fixed size arrays are encoded as per SOAP 1.1 one-dimensional array types. Multi-dimensional fixed size arrays are encoded by gSOAP as nested one-dimensional arrays in SOAP. Encoding of fixed size arrays supports partially transmitted and sparse array SOAP formats.

The decoding of (multi-dimensional) fixed-size arrays supports the SOAP multi-dimensional array format as well as partially transmitted and sparse array formats.

An example:

```
// Contents of header file "fixed.h":
struct Example
{
  float a[2][3];
};
```

This specifies a fixed-size array part of the **struct** Example. The encoding of array `a` is:

```
<a xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="float[][2]">
<SOAP-ENC:Array xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="float[3]"
<float xsi:type="float">...</float>
<float xsi:type="float">...</float>
<float xsi:type="float">...</float>
</SOAP-ENC:Array>
<SOAP-ENC:Array xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="float[3]"
<float xsi:type="float">...</float>
<float xsi:type="float">...</float>
<float xsi:type="float">...</float>
</SOAP-ENC:Array>
</a>
```

Caution: Any decoded parts of a (multi-dimensional) array that do not "fit" in the fixed size array are ignored by the deserializer.

9.9 Dynamic Arrays

As the name suggests, dynamic arrays are much more flexible than fixed-size arrays and dynamic arrays are better adaptabe to the SOAP encoding and decoding rules for arrays. In addition, a typical C application allocates a dynamic array using malloc, assigns the location to a pointer variable, and deallocates the array later with free. A typical C++ application allocates a dynamic array using new, assigns the location to a pointer variable, and deallocates the array later with

delete. Such dynamic allocations are flexible, but pose a problem for the serialization of data: how does the array serializer know the length of the array to be serialized given only a pointer to the sequence of elements? The application stores the size information somewhere. This information is crucial for the array serializer and has to be made explicitly known to the array serializer by packaging the pointer and array size information within a **struct** or **class**.

9.9.1 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 SOAP arrays is:

```
struct some_name
{
  Type *__ptr; // pointer to array
  int __size; // number of elements pointed to
  [[static const] int __offset [= ...];] // optional SOAP 1.1 array offset
  ... // anything that follows here will be ignored
};
```

where Type MUST be a type associated with an XML schema or MUST be a primitive type. If these conditions are not met, a vector-like XML (de)serialization is used (see Section 9.9.6). A primitive type can be used with or without a **typedef**. If the array elements are structs or classes, then the struct/class type names should have a namespace prefix for schema association, or they should be other (nested) dynamic arrays.

An alternative to a **struct** is to use a **class** with optional methods that MUST appear after the __ptr and __size fields:

```
class some_name
{
  public:
  Type *__ptr;
  int __size;
  [[static const] int __offset [= ...];]
  method1;
  method2;
  ... // any fields that follow will be ignored
};
```

To encode the data type as an array, the name of the **struct** or **class** SHOULD NOT have a namespace prefix, otherwise the data type will be encoded and decoded as a generic vector, see Section 9.9.6.

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.

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

9.9.2 Example

The following example header file specifies the XMethods Service Listing service getAllSOAPServices remote method and an array of SOAPService data structures:

```
// Contents of file "listing.h":
class ns3__SOAPService
{
  public:
  int ID;
  char *name;
  char *owner;
  char *description;
  char *homepageURL;
  char *endpoint;
  char *SOAPAction;
  char *methodNamespaceURI;
  char *serviceStatus;
  char *methodName;
  char *dateCreated;
  char *downloadURL;
  char *wsdlURL;
  char *instructions;
  char *contactEmail;
  char *serverImplementation;
};
class ServiceArray
{
  public:
  ns3__SOAPService *__ptr; // points to array elements
  int __size; // number of elements pointed to
  ServiceArray();
  ~ServiceArray();
  void print();
};
int ns__getAllSOAPServices(ServiceArray &return_);
```

An example client application:

```
#include "soapH.h" ...
// ServiceArray class method implementations:
ServiceArray::ServiceArray()
{
  __ptr = NULL;
  __size = 0;
}
ServiceArray::~ServiceArray()
{ // destruction handled by gSOAP
}
void ServiceArray::print()
{
  for (int i = 0; i ¡ __size; i++)
    cout << __ptr[i].name << ": " << __ptr[i].homepage << endl;
```

```
}
...
// Request a service listing and display results:
{
  struct soap soap;
  ServiceArray result;
  const char *endpoint = "www.xmethods.net:80/soap/servlet/rpcrouter";
  const char *action = "urn:xmethodsServicesManager#getAllSOAPServices";
  ...
  soap_init(&soap);
  soap_call_ns__getAllSOAPServices(&soap, endpoint, action, result);
  result.print();
  ...
  soap_destroy(&soap); // dealloc class instances
  soap_end(&soap); // dealloc deserialized data
  soap_done(&soap); // cleanup and detach soap struct
}
```

9.9.3 One-Dimensional Dynamic Arrays With Non-Zero Offset

The declaration of a dynamic array as described in 9.9 MAY include an **int** __offset field. When set to an integer value, the serializer of the dynamic array will use this field as the start index of the array and the SOAP array offset attribute will be used in the SOAP payload. Note that array offsets is a SOAP 1.1 specific feature which is not supported in SOAP 1.2.

For example, the following header file declares a mathematical Vector class, which is a dynamic array of floating point values with an index that starts at 1:

```
// Contents of file "vector.h":
typedef float xsd__float;
class Vector
{
  xsd__float *__ptr;
  int __size;
  int __offset;
  Vector();
  Vector(int n);
  float& operator[](int i);
}
```

The implementations of the Vector methods are:

```
Vector::Vector()
{
  __ptr = NULL;
  __size = 0;
  __offset = 1;
}
Vector::Vector(int n)
{
  __ptr = (float*)malloc(n*sizeof(float));
  __size = n;
```

```
  __offset = 1;
}
Vector::~Vector()
{
  if (__ptr)
    free(__ptr);
}
float& Vector::operator[](int i)
{
  return __ptr[i-__offset];
}
```

An example program fragment that serializes a vector of 3 elements:

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

The output is a partially transmitted array:

```
<vec xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:float[4]" SOAP-ENC:offset="[1]">
<item xsi:type="xsd:float">1.0</item>
<item xsi:type="xsd:float">2.0</item>
<item xsi:type="xsd:float">3.0</item>
</vec>
```

Note that the size of the encoded array is necessarily set to 4 and that the encoding omits the non-existent element at index 0.

The decoding of a dynamic array with an `__offset` field is more efficient than decoding a dynamic array without an `__offset` field, because the `__offset` field will be assigned the value of the `SOAP-ENC:offset` attribute instead of padding the initial part of the array with default values.

9.9.4 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:

```
// 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.

9.9.5 Multi-Dimensional Dynamic Arrays

The general form of the **struct** declaration for K-dimensional (K¿1) dynamic arrays is:

```
struct some_name
{
  Type *__ptr;
  int __size[K];
  int __offset[K];
  ... // anything that follows here will be ignored
};
```

where Type MUST be a type associated with an XML schema, which means that it must be a **typedef**ed type in case of a primitive type, or a **struct**/**class** name with a namespace prefix for schema association, or another dynamic array. If these conditions are not met, a generic vector XML (de)serialization is used (see Section 9.9.6).

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

```
class some_name
{
  public:
  Type *__ptr;
  int __size[K];
  int __offset[K];
  method1;
  method2;
  ... // any fields that follow will be ignored
};
```

In the above, K is a constant denoting the number of dimensions of the multi-dimensional array.

To encode the data type as an array, the name of the **struct** or **class** SHOULD NOT have a namespace prefix, otherwise the data type will be encoded and decoded as a generic vector, see Section 9.9.6.

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

For example, the following declaration specifies a matrix class:

```
typedef double xsd__double;
class Matrix
{
  public:
  xsd__double *__ptr;
```

```
  int __size[2];
  int __offset[2];
};
```

In contrast to the matrix class of Section 9.9.4 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.

9.9.6 Encoding XML Generics Containing Dynamic Arrays

XML "generics" extend the concept of a struct by allowing repetitions of elements within the struct. A simple generic is an array-like data structure with a repetition of one element. To achieve this, declare a dynamic array as a **struct** or **class** with a name that is qualified with a namespace prefix. SOAP arrays are declared without prefix.

For example:

```
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 generic with a list-like data structure. For example:

```
<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.

Multiple arrays can be used in a struct/class to support the concept of generics. Each array results in a repetition of elements in the struct/class. This is achieved with a **int** `__size` field in the struct/class where the next field (i.e. below the `__size` field) is a pointer type. The pointer type is

assumed to point to an array of values at run time. The __size field holds the number of values at run time. Multiple arrays can be embedded in a struct/class with __size fields that have a distinct names. To make the __size fields distinct, you can end them with a unique name suffix such as __sizeOfstrings, for example.

The general convention for embedding arrays is:

```
struct ns__SomeStruct
{
  ...
  int __sizename1; // number of elements pointed to
  Type1 *field1; // by this field
  ...
  int __sizename2; // number of elements pointed to
  Type2 *field2; // by this field
  ...
};
```

where name1 and name2 are identifiers used as a suffix to distinghuish the __size field. These names can be arbitrary and are not visible in XML.

For example, the following struct has two embedded arrays:

```
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:

```
<mycontact xsi:type="ns:Contact">
  <firstName>Joe</firstName>
  <lastName>Smith</lastName>
  <phoneNumber>5551112222</phoneNumber>
  <phoneNumber>5551234567</phoneNumber>
  <phoneNumber>5552348901</phoneNumber>
  <emailAddress>Joe.Smith@mail.com</emailAddress>
  <emailAddress>Joe@Smith.com</emailAddress>
  <socSecNumber>999999999</socSecNumber>
</mycontact>
```

9.9.7 STL Containers

gSOAP supports the STL containers std::deque, std::list, std::set, and std::vector.

STL containers can only be used within classes to declare members that contain multiple values. This is somewhat similar to the embedding of arrays in structs in C as explained in Section 9.9.6, but the STL container approach is more flexible.

You need to import stldeque.h, stllist.h, stlset.h, or stlvector.h to enable std::deque, std::list, std::set, and std::vector (de)serialization. Here is an example:

```
#import "stlstring.h"
#import "stlvector.h"
class ns__myClass
{ public:
  std::vector<int> *number;
  std::vector<xsd__string> *name;
  ...
};
```

The use of pointer members is not required but advised. The reason is that interoperability with other SOAP toolkits may lead to copying of ns__myClass instances at run time when (de)serializing multi-referenced data. When a copy is made, certain parts of the containers will be shared between the copies which could lead to disaster when the classes with their containers are deallocated. Another way to avoid this is to declare class ns__myClass within other data types via a pointer. (Interoperability between gSOAP clients and services does not lead to copying.)

The XML schema that corresponds to the ns__myClass type is

```
<complexType name="myClass">
  <sequence>
    <element name="number" type="xsd:int" minOccurs="1" maxOccurs="unbounded"/>
    <element name="name" type="xsd:string" minOccurs="1" maxOccurs="unbounded"/>
    ...
  </sequence>
</complexType>
```

You can specify the minOccurs and maxOccurs values as explained in Section 13.2.

You can also implement your own containers similar to STL containers. The containers must be class templates with the following members:

- myContainer<T>::iterator and myContainer<T>::const_iterator;
- myContainer<T>::iterator myContainer<T>::insert(myContainer<T>::iterator, myContainer<T>::value_type&);
- myContainer<T>::const_iterator begin();
- myContainer<T>::const_iterator end();

The myContainer<T>::iterator should have a dereference operator to access the container's elements. The dereference operator is used by gSOAP to send a sequence of XML element values. The insert method can be used as a setter method. gSOAP reads a sequence of XML element values and inserts whem in the container via this method. To enable you containers, add the following line to your gSOAP header file:

```
#include "myContainer.h"
templare ¡class T¿ class myContainer;
```

where myContainer is the name of your container class. The container class should not be defined in the gSOAP header file. You should define it in separate sources ("myContainer.h") and link those to your application. Use option -i with the gSOAP compiler to handle #include "myContainer".

Caution: when parsing XML content the container elements may not be stored in the same order given in the XML content. When gSOAP parses XML it uses the insert container methods to store elements one by one. However, element content that is "forwarded" with `href` attributes will be appended to the container. Forwarding can take place with multi-referenced data that is referred to from the main part of the SOAP 1.1 XML message to the independent elements that carry `id`s. Therefore, your application should not rely on the preservation of the order of elements in a container.

9.9.8 Polymorphic Dynamic Arrays and Lists

Polymorphic arrays (arrays of polymorphic element types) can be encoded when declared as an array of pointers to class instances. and lists. For example:

```
class ns__Object
{
  public:
  ...
};
class ns__Data: public ns__Object
{
  public:
  ...
};
class ArrayOfObject
{
  public:
  ns__Object **__ptr; // pointer to array of pointers to Objects
  int __size; // number of Objects pointed to
  int __offset; // optional SOAP 1.1 array offset
};
```

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.

9.9.9 How to Change the Tag Names of the Elements of a SOAP Array or List

The __ptr field in a **struct** or **class** declaration of a dynamic array may have an optional suffix part that describes the name of the tags of the SOAP array XML elements. The suffix is part of the field name:

```
Type *__ptrarray_elt_name
```

The suffix describes the tag name to be used for all array elements. The usual identifier to XML translations apply, see Section 8.3. The default XML element tag name for array elements is `item` (which corresponds to the use of field name __ptritem).

Consider for example:

```
struct ArrayOfstring
{
  xsd__string *__ptrstring;    int __size; };
```

The array is serialized as:

```
<array xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[2]">
<string xsi:type="xsd:string">Hello</string>
<string xsi:type="xsd:string">World</string>
</array>
```

SOAP 1.1 and 1.2 do not require the use of a specific tag name for array elements. gSOAP will deserialize a SOAP array while ignoring the tag names. Certain XML schemas used in doc/literal encoding may require the declaration of arrray element tag names.

9.10 Base64Binary XML Schema Type Encoding

The `base64Binary` XML schema type is a special form of dynamic array declared with a pointer (`__ptr`) to an **unsigned char** array.

For example using a **struct**:

```
struct xsd__base64Binary
{
  unsigned char *__ptr;
  int __size;
};
```

Or with a **class**:

```
class xsd__base64Binary
{
  public:
  unsigned char *__ptr;
  int __size;
};
```

When compiled by the gSOAP 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:

```
struct SOAP_ENC__base64
{
  unsigned char *__ptr;
  int __size;
};
```

Or with a **class**:

```
class SOAP_ENC__base64
{
  unsigned char *__ptr;
  int __size;
};
```

When compiled by the gSOAP 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:

```
class xsd__base64Binary
{
  public:
  unsigned char *__ptr;
  int __size;
  xsd__base64Binary(); // Constructor
  xsd__base64Binary(struct soap *soap, int n); // Constructor
  ~xsd__base64Binary(); // Destructor
  unsigned char *location(); // returns the memory location
  int size(); // returns the number of bytes
};
```

Here are example method implementations:

```
xsd__base64Binary::xsd__base64Binary()
{
  __ptr = NULL;
  __size = 0;
}
xsd__base64Binary::xsd__base64Binary(struct soap *soap, int n)
{
  __ptr = (unsigned char*)soap_malloc(soap, n);
  __size = n;
}
xsd__base64Binary::~xsd__base64Binary()
{ }
unsigned char *xsd__base64Binary::location()
{
  return __ptr;
}
int xsd__base64Binary::size()
{
  return __size;
}
```

The following example in C/C++ reads from a raw image file and encodes the image in SOAP using the `base64Binary` type:

```
...
FILE *fd = fopen("image.jpg", "r");
```

```
xsd__base64Binary image(&soap, filesize(fd));
fread(image.location(), image.size(), 1, fd);
fclose(fd);
soap_begin(&soap);
image.soap_serialize(&soap);
image.soap_put(&soap, "jpegimage", NULL);
soap_end(&soap);
...
```

where filesize is a function that returns the size of a file given a file descriptor.

Reading the `xsd:base64Binary` encoded image.

```
...
xsd__base64Binary image;
soap_begin(&soap);
image.get(&soap, "jpegimage");
soap_end(&soap);
...
```

The **struct** or **class** name soap_enc__base64 should be used for `SOAP-ENC:base64` schema type instead of xsd__base64Binary.

9.11 hexBinary XML Schema Type Encoding

The `hexBinary` XML schema type is a special form of dynamic array declared with the name xsd__hexBinary and a pointer (__ptr) to an **unsigned char** array.

For example, using a **struct**:

```
struct xsd__hexBinary
{
  unsigned char *__ptr;
  int __size;
};
```

Or using a **class**:

```
class xsd__hexBinary
{
  public:
  unsigned char *__ptr;
  int __size;
};
```

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

9.12 Literal XML Encoding Style

gSOAP supports SOAP RPC encoding by default. Literal encoding is optional. Just as with SOAP RPC encoding, literal encoding requires the XML schema of the message data to be provided e.g. in

WSDL in order for the gSOAP compiler to generate the (de)serialization routines. Alternatively, the optional DOM parser (dom.c and dom++.cpp) can be used to handle generic XML or arbitrary XML documents can be (de)serialized into regular C strings or wide character strings (wchar_t*) by gSOAP (see Section 9.12.1).

gSOAP supports literal SOAP encoding either manually by setting the value of soap.encodingStyle and by using the flag SOAP_XML_TREE for output mode in your code, or automatically by using a gSOAP directive in the header file. The directive to enable doc/lit for the entire service is

```
//gsoap ns service encoding: literal
```

To enable literal encoding for particular service methods, use:

```
//gsoap ns service method-encoding: myMethod literal
int ns__myMethod(...)
```

The encodings can also be controlled manually as follows. In the case of doc/literal encoding, the `SOAP-ENV:encodingStyle` attribute must be absent in SOAP/XML messages. To do this, set soap.encodingStyle=NULL. Finally, rpc/lit is a limited form of serialization and do not support graphs serializations. So setting the flag SOAP_XML_TREE will produce tree-structured output preventing multi-reference data. Note that cyclic data will crash the serializer because of this setting. Also polymorphic data may cause deserialization problems due to the absense of type information in the SOAP payload (which makes us wonder why doc/literal is the default in some SOAP toolkits).

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

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

Note that none of the data types need to be namespace qualified using namespace prefixes.

```
//gsoap ns service name: localtime
//gsoap ns service encoding: literal
//gsoap ns service namespace: http://alethea.net/webservices/
int ns__LocalTimeByZipCode(char *ZipCode, char **LocalTimeByZipCodeResult);
```

In this case, the method name requires to be associated with a schema through a namespace prefix, e.g. ns is used in this example. See Section 13.2 for more details on gSOAP directives. With these directives, the gSOAP compiler generates client and server sources with the specified settings.

The example client program is:

```
#include "soapH.h"
#include "localtime.nsmap" // include generated map file
int main()
{
  struct soap soap;
  char *t;
  soap_init(&soap);
  if (soap_call_ns__LocalTimeByZipCode(&soap, "http://alethea.net/webservices/LocalTime.asmx",
```

```
"http://alethea.net/webservices/LocalTimeByZipCode", "32306", &t))
    soap_print_fault(&soap, stderr);
  else
    printf("Time = %s\n", t);
  return 0;
}
```

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

```
#include "soapH.h"
#include "localtime.nsmap" // include generated map file
int main()
{
  struct soap soap;
  char *t;
  soap_init(&soap);
  soap.encodingStyle = NULL; // don't use SOAP encoding
  soap_set_omode(&soap, SOAP_XML_TREE);" // don't produce multi-ref data (but can accept)
  if (soap_call_ns__LocalTimeByZipCode(&soap, "http://alethea.net/webservices/LocalTime.asmx",
"http://alethea.net/webservices/LocalTimeByZipCode", "32306", &t))
    soap_print_fault(&soap, stderr);
  else
    printf("Time = %s\n", t);
  return 0;
}
```

The SOAP request is:

```
POST /webservices/LocalTime.asmx HTTP/1.0
Host:  alethea.net
Content-Type:  text/xml; charset=utf-8
Content-Length:  479
SOAPAction:  "http://alethea.net/webservices/LocalTimeByZipCode"

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <SOAP-ENV:Body>
    <LocalTimeByZipCode xmlns="http://alethea.net/webservices/">
<ZipCode>32306</ZipCode></LocalTimeByZipCode>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

9.12.1 Serializing and Deserializing Mixed Content XML With Strings

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

```
typedef char *XML;
```

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

```
typedef wchar_t *XML;
```

Note: only one of the two storage formats can be used. The differences between the use of regular strings versus wide character strings for XML documents are:

- Regular strings for XML documents MUST hold UTF-8 encoded XML documents. That is, the string MUST contain the proper UTF-8 encoding to exchange the XML document in SOAP messages.
- Wide character strings for XML documents SHOULD NOT hold UTF-8 encoded XML documents. Instead, the UTF-8 translation is done automatically by the gSOAP runtime marshalling routines.

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:

```
typedef char *XML;
ns__GetDocument(XML m__XMLDoc, XML &m__XMLDoc_);
```

The ns__Document is essentially a **struct** that forms the root of the XML document. The use of the underscore in the ns__Document response part of the message avoids the name clash between the **struct**s. Assuming that the namespace mapping table contains the binding of ns to `http://my.org/` and the binding of m to `http://my.org/mydoc.xsd`, the XML message is:

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

Important: the literal XML encoding style MUST be specified by setting soap.encodingStyle, where soap is a variable that contains the current runtime environment. For example, to specify no constraints on the encoding style (which is typical) use NULL:

```
struct soap soap;
soap_init(&soap);
soap.encodingStyle = NULL;
```

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

For interoperability with Apache SOAP, use

```
struct soap soap;
soap_init(&soap);
soap.encodingStyle = "http://xml.apache.org/xml-soap/literalxml";
```

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

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

10 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:

```
struct SOAP_ENV__Fault
{
  char *faultcode;
  char *faultstring;
  char *faultactor;
  char *detail;
  struct SOAP_ENV__Code *SOAP_ENV__Code;
  char *SOAP_ENV__Reason;
  char *SOAP_ENV__Detail;
};
struct SOAP_ENV__Code
{
  char *SOAP_ENV__Value;
  char *SOAP_ENV__Node;
  char *SOAP_ENV__Role;
};
```

The first four fields are SOAP 1.1 specific. The last three fields are SOAP 1.2 specific. 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 in the header file input to the gSOAP compiler to replace the built-in data structure. For example:

```
struct SOAP_ENV__Fault
{
  char *faultcode; // MUST be string
  char *faultstring; // MUST be string
  char *faultactor;
```

```
  Detail *detail; // new detail field
  struct SOAP_ENV_Code *SOAP_ENV__Code; // MUST be a SOAP_ENV__Code struct defined
below
  char *SOAP_ENV__Reason; // MUST be string
  Detail SOAP_ENV__Detail; // new SOAP 1.2 detail field
}; struct SOAP_ENV__Code
{
  char *SOAP_ENV__Value; // MUST be string
  char *SOAP_ENV__Node;
  char *SOAP_ENV__Role;
};
```

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

When the skeleton of a remote method returns an error (see Section 8.2), then soap.fault contains the SOAP Fault data at the receiving side (client).

When a remote method wants to raise an exception, it does so by assigning the fault field of the current reference to the runtime environment with appropriate data associated with the exception and by returning the error SOAP_FAULT. For example:

```
soap_receiver_fault(soap, "Stack dump", NULL);
if (soap-¿version == 2)
  soap->fault->SOAP_ENV__Detail = sp; // SOAP 1.2: point to stack
else
  soap->fault->detail = sp; // SOAP 1.1: point to stack
return SOAP_Fault; // return from remote method call
```

When soap_fault allocates a fault struct, this data is removed with the soap_end call (or soap_dealloc). Note that the soap_receiver_fault function is called to allocate the fault struct and set the fault string and detail fields, i.e. soap_receiver_fault(soap, "Stack dump", NULL). The advantage is that this is independent of SOAP 1.1 and SOAP 1.2. However, setting the custom detail fields requires inspecting the SOAP version used, using the soap-¿version attribute which is 1 for SOAP 1.1 and 2 for SOAP 1.2.

Each remote method implementation in a service application can return a SOAP Fault upon an exception by returning an error code, see Section 6.2.1 for details and an example. In addition, a SOAP Fault can be returned by a service application through calling the soap_send_fault function. This is useful in case the initialization of the application fails, as illustrated in the example below:

```
int main()
{
  struct soap soap;
  soap_init(&soap);
  some initialization code
  if (initialization failed)
  {
    soap.error = soap_receiver_fault(&soap, "Init failed", "..."); // set the error condition (SOAP_FAULT)
    soap_send_fault(&soap); // Send SOAP Fault to client
    return 0; // Terminate
  }
}
```

11 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:

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

which declares and 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 can be added to the header file input to the gSOAP compiler. A **class** for the SOAP Header data structure can be used instead of a **struct**.

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

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

with client-side code:

```
  struct soap soap;
  soap_init(&soap);
...
soap.header = NULL; // do not use a SOAP Header for the request (as set with soap_init)
soap.actor = NULL; // do not use an actor (receiver is actor)
soap_call_method(&soap, ...);
if (soap.header) // a SOAP Header was received
  cout << soap.header->t__transaction;
// Can reset, modify, or set soap.header here before next call
soap_call_method(&soap, ...); // reuse the SOAP Header of the service response for the request
...
```

The SOAP Web service response can include a SOAP Header with a transaction number that the client is supposed to use for the next remote method invocation to the service. Therefore, the next request includes a transaction number:

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

This is just an example and the transaction control is not a feature of SOAP but can be added on by the application layer to implement stateful transactions between clients and services. At the

client side, the soap.actor attribute can be set to indicate the recipient of the header (the SOAP `SOAP-ENV:actor` attribute).

A Web service can read and set the SOAP Header as follows:

```
int main()
{
  struct soap soap;
  soap.actor = NULL; // use this to accept all headers (default)
  soap.actor = "http://some/actor"; // accept headers destined for "http://some/actor" only
  soap_serve(&soap);
}
...
int method(struct soap *soap, ...)
{
  if (soap->header) // a Header was received
    ... = soap->header->t__transaction;
  else
    soap->header = soap_malloc(sizeof(struct SOAP_ENV__Header)); // alloc new header
...     soap->header->t__transaction = ...;
  return SOAP_OK;
}
```

See Section 13.2 on how to generate WSDL with the proper method-to-header-part bindings.

The `SOAP-ENV:mustUnderstand` attribute indicates the requirement that the recipient of the SOAP Header (who must correspond to the `SOAP-ENV:actor` attribute when present or when the attribute has the value `SOAP-ENV:actor="http://schemas.xmlsoap.org/soap/actor/next"`) MUST handle the Header part that carries the attribute. gSOAP handles this automatically on the background. However, an application still needs to inspect the header part's value and handle it appropriately. If a 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" attrbute. For example:

```
struct SOAP_ENV__Header
{
  char *t__transaction;
  mustUnderstand char *t__authentication;
};
```

When both fields are set and soap.actor="http://some/actor" then the message contains:

```
<SOAP-ENV:Envelope ...>
<SOAP-ENV:Header>
<t:transaction>5</t:transaction>
<t:authentication SOAP-ENV:actor="http://some/actor" SOAP-ENV:mustUnderstand="1">XX
</t:authentication>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
...
```

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

12 DIME Attachment Processing

gSOAP can transmit binary data with DIME attachments with or without streaming. With DIME output streaming, the binary data is retrieved from an application's data source at run time in parts without storing the entire content. With DIME input streaming, the binary data will be handed to the application in parts. DIME streaming is implemented with function callbacks. See Section 12.2 for more details.

12.1 Non-Streaming DIME

Without streaming, the binary data is stored in augmented `xsd:base64Binary` and `xsd:hexBinary` structs/classes. These structs/classes have three additional fields: an id field for attachment referencing (typically a content id (CID) or UUID), a type field to specify the MIME type of the binary data, and an options field to piggy-back additional information with a DIME attachment. DIME attachment support is fully automatic, which means that gSOAP will test for the presence of attachments at run time and use SOAP in DIME accordingly.

A `xsd:base64Binary` type with DIME attachment support is declared by

```
struct xsd__base64Binary
{
  unsigned char *__ptr;
  int __size;
  char *id;
  char *type;
  char *options;
};
```

The specification order of the fields is significant. In addition, no other fields or methods may be declared before any of these fields in the struct/class, but additional fields and methods can appear after the field declarations. A xsd:hexBinary declaration is similar. When the id field and/or type field is non-NULL during run-time serialization of the data, DIME attachment transmission is used for the *entire* SOAP message, as per SOAP in DIME specifications. When only the type field is set, gSOAP will assign a default DIME id to the attachment (see also below). The options field is an optional string with a special layout: the first two bytes are reserved for the option type, the next two bytes store the size of the option data, followed by the option data. The function

```
char *soap_dime_option(struct soap *soap, unsigned short type, const char *option)
```

returns a string with this encoding. For example

```
struct xsd__base64Binary image;
image.__ptr = ...;
image.__size = ...;
image.id = "uuid:09233523-345b-4351-b623-5dsf35sgs5d6";
image.type = "image/jpeg";
image.options = soap_dime_option(soap, 0, "My wedding picture");
```

When receiving DIME attachments, the fields will be set according to the DIME attachment content. If binary data is received without attachments, the id, type, options fields are all NULL. Note that SOAP messages may contain binary data that references external resources not provided as attachments. In that case, the __ptr field is NULL and the id field refers to the external data source. Non-augmented binary data types `xsd:base64Binary` and `xsd:hexBinary` can be used to receive DIME attachments, but the id, type, and options information are absent. Also DIME attachments can be received and stored in strings, but not send from strings.

When necessary, the `xsd:base64Binary` schema type and its attachment-based counterpart can be specified with class inheritance. For example:

```
class xsd__base64Binary
{
  unsigned char *__ptr;
  int __size;
};
class xsd__base64Binary_ : xsd__base64Binary
{
  char *id;
  char *type;
  char *options;
};
```

The dime_id_format attribute of the current gSOAP run-time environment can be set to the default format of DIME id fields. The format string MUST contain a %d format specifier (or any other **int**-based format specifier). The value of this specifier is a non-negative integer, with zero being the value of the DIME attachment id for the SOAP message. For example,

```
struct soap soap;
soap_init(&soap);
soap.dime_id_format = "uuid:09233523-345b-4351-b623-5dsf35sgs5d6-%x";
```

As a result, all attachments with a NULL id field will use a gSOAP-generated id value based on the format string.

12.2 Streaming DIME

Streaming DIME is achieved with callback functions to fetch and store data during transmission. Three function callbacks for streaming DIME output and three callbacks for streaming DIME input are available. These callbacks are used to access application data resources.

Callback (function pointer)
void *(*soap.fdimereadopen)(**struct** soap *soap, **void** *handle, **const char** *id, **const char** *type, **const char** *options) Called by the gSOAP run-time DIME attachment sender to start reading from a (binary) data source for outbound transmission. The content will be read from the application's data source in chunks using the fdimeread callback and streamed into the SOAP/XML/DIME output stream. The handle contains the value of the __ptr field of an attachment struct/class, which could be a pointer to specific information such as a file descriptor or a pointer to a string to be passed to this callback. Both __ptr and __size fields should have been set by the application prior to the serialization of the content. The id, type, and options arguments are the DIME id, type, and options, respectively. The callback should return handle, or another pointer value which will be passed as a handle to fdimeread and fdimereadclose. The callback should return NULL and set soap->error when an error occurred. The callback should return NULL (and not set soap->error) when this particular DIME attachment is not to be streamed.
size_t (*soap.fdimeread)(**struct** soap *soap, **void** *handle, **char** *buf, size_t len) Called by the gSOAP run-time DIME attachment sender to read more data from a (binary) data source for streaming into the output stream. The handle contains the value returned by the fdimereadopen callback. The buf argument is the buffer of length len into which a chunk of data should be stored. The actual amount of data stored in the buffer may be less than len and this amount should be returned by the application. A return value of 0 indicates an error (the callback may set soap->errnum to errno). The __size field of the attachment struct/class should have been set by the application prior to the serialization of the content. The value of __size indicates the total size of the content to be transmitted. When the __size is zero then DIME chunked transfers can be used under certain circumstances to stream content without prior determination of attachment size, see Section 12.3 below.
void(*soap.fdimereadclose)(**struct** soap *soap, **void** *handle) Called by the gSOAP run-time DIME attachment sender at the end of the streaming process to close the data source. The handle contains the value returned by the fdimereadopen callback. The fdimewriteclose callback is called after successfully transmitting the data or when an error occurred.
void *(*soap.fdimewriteopen)(**struct** soap *soap, **const char** *id, **const char** *type, **const char** *options) Called by the gSOAP run-time DIME attachment receiver to start writing an inbound DIME attachment to an application's data store. The content is streamed into an application data store through multiple fdimewrite calls from the gSOAP attachment receiver. The id, type, and options arguments are the DIME id, type, and options respectively. The callback should return a handle which is passed to the fdimewrite and fdimewriteclose callbacks. The __ptr field of the attachment struct/class is set to the value of this handle. The __size field is set to the total size of the attachment after receving the entire content. The size is unknown in advance because DIME attachments may be chunked.
int (*soap.fdimewrite)(**struct** soap *soap, **void** *handle, **const char** *buf, size_t len) Called by the gSOAP run-time DIME attachment receiver to write part of an inbound DIME attachment to an application's data store. The handle contains the value returned by the fdimewriteopen callback. The buf argument contains the data of length len. The callback should return a gSOAP error code (e.g. SOAP_OK when no error occurred).
void(*soap.fdimewriteclose)(**struct** soap *soap, **void** *handle) Called by the gSOAP run-time DIME attachment receiver at the end of the streaming process to close the data store. The fdimewriteclose callback is called after successfully receiving the data or when an error occurred. The handle contains the value returned by the fdimewriteopen callback.

In addition, a **void***user field in the **struct** soap data structure is available to pass user-defined data to the callbacks. This way, you can set soap.user to point to application data that the callbacks need such as a file name for example.

The following example illustrates the client-side initialization of an image attachment struct to stream a file into a DIME attachment:

```
int main()
{
  struct soap soap;
  struct xsd__base64Binary image;
  FILE *fd;
  struct stat sb;
  soap_init(&soap);
  if (!fstat(fileno(fd), &sb) && sb.st_size > 0)
  { // because we can get the length of the file, we can stream it
    soap.fdimereadopen = dime_read_open;
    soap.fdimereadclose = dime_read_close;
    soap.fdimeread = dime_read;
    image.__ptr = (unsigned char*)fd; // must set to non-NULL (this is our fd handle which we
need in the callbacks)
    image.__size = sb.st_size; // must set size
  }
  else
  { // don't know the size, so buffer it
    size_t i;
    int c;
    image.__ptr = (unsigned char*)soap_malloc(&soap, MAX_FILE_SIZE);
    for (i = 0; i < MAX_FILE_SIZE; i++)
    {
      if ((c = fgetc(fd)) == EOF)
        break;
      image.__ptr[i] = c;
    }
    fclose(fd);
    image.__size = i;
  }
  image.type = "image/jpeg";
  image.options = soap_dime_option(&soap, 0, "My picture");
  soap_call_ns__method(&soap, ...);
  ...
}
void *dime_read_open(struct soap *soap, void *handle, const char *id, const char *type, const
char *options)
{ return handle;
}
void dime_read_close(struct soap *soap, void *handle)
{ fclose((FILE*)handle);
}
size_t dime_read(struct soap *soap, void *handle, char *buf, size_t len)
{ return fread(buf, 1, len, (FILE*)handle);
}
```

The following example illustrates the streaming of a DIME attachment into a file by a client:

```
int main()
{ struct soap soap;
```

```
  soap_init(&soap);
  soap.fdimewriteopen = dime_write_open;
  soap.fdimewriteclose = dime_write_close;
  soap.fdimewrite = dime_write;
  soap_call_ns__method(&soap, ...);
  ...
}
void *dime_write_open(struct soap *soap, const char *id, const char *type, const char *options)
{
  FILE *handle = fopen("somefile", "wb");
  if (!handle)
  {
    soap->error = SOAP_EOF;
    soap->errnum = errno; // get reason
  }
  return (void*)handle;
}
void dime_write_close(struct soap *soap, void *handle)
{ fclose((FILE*)handle);
}
int dime_write(struct soap *soap, void *handle, const char *buf, size_t len)
{
  size_t nwritten;
  while (len)
  {
    nwritten = fwrite(buf, 1, len, (FILE*)handle);
    if (!nwritten)
    {
      soap->errnum = errno; // get reason
      return SOAP_EOF;
    }
    len -= nwritten;
    buf += nwritten;
  }
  return SOAP_OK;
}
```

Note that compression can be used with DIME to compress the entire message. However, compression requires buffering to determine the HTTP content length header thereby canceling the benefits of streaming DIME. Use chunked HTTP (with the output-mode SOAP_IO_CHUNK flag) with compression and streaming DIME. However, compression introduces a significant processing overhead that should not be ignored.

12.3 Streaming Chunked DIME

gSOAP automatically handles inbound chunked DIME attachments (streaming or non-streaming). To transmit outbound DIME attachments, the attachment sizes MUST be determined in advance to calculate HTTP message length required to stream DIME over HTTP. However, gSOAP also supports the transmission of outbound chunked DIME attachments without prior determination of DIME attachment sizes when certain conditions are met. These conditions require either non-HTTP transport (use the output-mode SOAP_ENC_XML flag), or chunked HTTP transport (use the

output-mode SOAP_IO_CHUNK flag). You can also use the SOAP_IO_STORE flag (which is also used automatically with compression to determine the HTTP content length header) but that cancels the benefits of streaming DIME.

To stream chunked DIME, set the __size field of an attachment to zero and enable HTTP chunking. The DIME fdimeread callback then fetches data in chunks and it is important to fill the entire buffer unless the end of the data has been reached and the last chunk is to be send. That is, fdimeread should return the value of the last len parameter and fill the entire buffer buf for all chunks except the last.

13 Advanced Features

13.1 Internationalization

gSOAP uses regular strings by default. Regular strings cannot be used to hold UCS characters outside of the ASCII character range [1,255]. gSOAP can handle wide-character content in two ways. First, applications can utilize wide-character strings (wchar_t*) instead of regular strings to store wide-character content. For example, the `xsd:string` string schema type can be declared as a wide-character string and used subsequently:

```
typedef wchar_t *xsd__string;
...
int ns__myMethod(xsd__string input, xsd__string *output);
```

Second, regular strings can be used to hold wide-character content in UTF-8 format. This is accomplished with the SOAP_C_UTFSTRING flag, see Section 7.12. With this flag set, gSOAP will deserialize XML into regular strings in UTF-8 format. An application is responsible for filling regular strings with UTF-8 content to serialize XML.

Both regular strings and wide-character strings can be used within an application. For example, the following header file declaration introduces two string schema types:

```
typedef wchar_t *xsd__string;
typedef char *xsd__string_; // trailing '_' avoids name clash
...
int ns__myMethod(xsd__string input, xsd__string_ *output);
```

The input string parameter is a wide-character string and the output string parameter is a regular string. The regular string has ASCII content unless the SOAP_C_UTFSTRING flag is set. With this flag, the string has UTF-8 content. Note that ASCII format only supports characters in the range [1,255].

Please consult the UTF-8 specification for details on the UTF-8 format.

13.2 Customizing the WSDL and Namespace Mapping Table File Contents With gSOAP Directives

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 6.2.7 and 8.4). In addition, the sample SOAP/XML request and response files generated by the compiler are valid provided that XML schema namespace information is added to the header file with directives so that the gSOAP compiler can produce example SOAP/XML messages that are correctly namespace qualified. These compiler directive 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). The name may be followed by text up to the end of the line which is incorporated into the WSDL service documentation. Alternatively, the service documentation can be provided with the directive below.

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

//gsoap *namespace-prefix* service documentation: *text*

where *namespace-prefix* is a namespace prefix used by identifiers in the header file and *text* is the documentation text up to the end of the line. The text is incorporated into the WSDL service documentation.

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

//gsoap *namespace-prefix* service portType: *portType*

where *namespace-prefix* is a namespace prefix used by identifiers in the header file and *portType* is the portType name of the WSDL service portType.

To specify the location (or port endpoint) of a Web Service in the header file, use:

//gsoap *namespace-prefix* service location: *URL*

or alternatively

//gsoap *namespace-prefix* service port: *URL*

where *URL* is the location of the Web Service (only required to create new Web Services). The *URL* specifies the path to the service executable (so *URL*/*service-executable* is the actual location of the executable when declared).

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

//gsoap *namespace-prefix* service executable: *executable-name*

where *executable-name* is the name of the executable of the Web Service.

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

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

or when the `SOAP-ENV:encodingStyle` attribute is different from the SOAP 1.1/1.2 encoding style:

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

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

```
//gsoap namespace-prefix service namespace: namespace-URI
```

where *namespace-URI* is the URI associated with the namespace prefix.

In addition, the schema namespace URI can be specified in the header file:

```
//gsoap namespace-prefix schema namespace: namespace-URI
```

where *namespace-URI* is the schema URI associated with the namespace prefix. If present, it defines the schema-part of the generated WSDL file and the URI in the namespace mapping table. This declaration is useful when the service declares its own data types that need to be associated with a namespace. Furthermore, the header file for client applications do not need the full service details and the specification of the schema namespaces for namespace prefixes suffices.

The directive above specifies a new schema and the gSOAP compiler generates a schema files (.xsd) file for the schema. An existing schema namespace URI can be imported with:

```
//gsoap namespace-prefix schema import: namespace-URI
```

where *namespace-URI* is the schema URI associated with the namespace prefix. gSOAP does not produce XML schema files for imported schemas and imports the schema namespaces in the generated WSDL file.

A schema namespace URI can be imported from a location with:

```
//gsoap namespace-prefix schema namespace: namespace-URI
//gsoap namespace-prefix schema import: schema-location
```

To document a method, use:

```
//gsoap namespace-prefix service method-documentation: method-name //text
```

where *method-name* is the unqualified name of the method and *text* is a line of text terminated by a newline. Do not use any XML reserved characters in *text* such as < and >. Use XML and XHTML markup instead. For example:

```
//gsoap ns service method-documentation: getQuote This method returns a stock quote
int ns__getQuote(char *symbol, float &_result);
```

To specify the SOAPAction for a method, use:

```
//gsoap namespace-prefix service method-action: method-name action
```

where *method-name* is the unqualified name of the method and *action* is a quoted or non-quoted string (spaces and blanks are not allowed). For example:

```
//gsoap ns service method-action: getQuote ""
int ns__getQuote(char *symbol, float &_result);
```

When literal encoding is required for a particular service method, use:

```
//gsoap namespace-prefix service method-encoding: method-name literal
```

or when the `SOAP-ENV:encodingStyle` attribute is different from the SOAP 1.1/1.2 encoding style, use:

```
//gsoap namespace-prefix service method-encoding: method-name encoding-style
```

When header processing is required, each method declared in the WSDL should provide a binding to the parts of the header that may appear as part of a method request message. Such a binding is given by:

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

For example:

```
struct SOAP_ENV__Header
{
  char *h__transaction;
  struct UserAuth *h__authentication;
};
```

Suppose method `ns__login` uses both header parts (at most), then this is declared as:

```
//gsoap ns service method-header-part: login 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:

```
//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:

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

Similarly, to specify the header parts for the method output (method response message), use:

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

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

(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:

```
//gsoap ns1 service namespace: urn:xmethods-delayed-quotes
int ns1__getQuote(char *symbol, float &result);

//gsoap ns2 service namespace: urn:xmethods-CurrencyExchange
int ns2__getRate(char *country1, char *country2, float &result);

//gsoap ns3 service name: quotex
//gsoap ns3 service location: http://www.cs.fsu.edu/~engelen
//gsoap ns3 service namespace: urn:quotex
int ns3__getQuote(char *symbol, char *country, float &result);
```

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:

```
soapcpp2 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 http://tempuri.org).

13.3 How to Specify minOccurs and maxOccurs

By default, gSOAP generates WSDL and schemas with `minOccurs=1` and `maxOccurs=1` for non-array types, and `minOccurs=0` and `maxOccurs=unbounded` for array types. The `minOccurs` and `maxOccurs` attribute values of fields in **struct** and **class** types are specified as

Type fieldname [minOccurs[:maxOccurs]] [= value]

The minOccurs and maxOccurs values must be integer literals.

For example

```
struct ns__MyRecord
{
  int n;
  int m 0;
  int __size 0:10;
  int *item;
}
```

gSOAP generates:

```
<complexType name="MyRecord">
  <all>
    <element name="n" type="xsd:int" minOccurs="1" maxOccurs="1"/>
    <element name="m" type="xsd:int" minOccurs="0" maxOccurs="1"/>
    <element name="item" type="xsd:int" minOccurs="0" maxOccurs="10"/>
  </all>
</complexType>
```

13.4 How to Specify a SimpleType and Pattern

You can define a schema simpleType with a **typedef**, which takes a base primitive type and defines a new simpleType. For example:

```
typedef int time__seconds;
```

This defines the following schema type in time.xsd:

```
<simpleType name="seconds">
  <restriction base="xsd:int"/>
</simpleType>
```

To associate a pattern with a simpleType, you can define a simpleType with a **typedef** and a pattern string:

```
typedef int time__seconds "[1-5]?[0-9]|60";
```

This defines the following schema type in time.xsd:

```
<simpleType name="seconds">
  <restriction base="xsd:int">
    <pattern value="[1-5]?[0-9]|60"/>
  </restriction base="xsd:int"/>
</simpleType>
```

The pattern string MUST contain a valid regular expression.

13.5 Transient Data Types

There are situations when certain data types have to be ignored by gSOAP for the compilation of (de)marshalling routines. For example, in certain cases only a few members of a class or struct need not be (de)serialized, or the base class of a derived class should not be (de)serialized. Certain built-in classes such as ostream cannot be (de)serialized. Data parts that should be kept invisible to gSOAP are called "transient". Transient data types and transient struct/class members are declared with the **extern** keyword or are declared within [and] blocks in the header file. The **extern** keyword has a special meaning to the gSOAP compiler and won't affect the generated codes. The special [and] block construct can be used with data type declarations and within **struct** and **class** declarations. The use of **extern** or [ ] achieve the same effect, but [ ] may be more convenient to encapsulate transient types in a larger part of the header file. The use of **extern** with **typedef** is reserved for the declaration of user-defined external (de)serializers for data types, see Section 13.7.

First example:

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

Second example:

```
[
  class myBase // base class need not be (de)serialized
  { ... };
]
class ns__myDerived : myBase
{ ... };
```

Third example:

```
[ typedef int transientInt; ]
class ns__myClass
{
  int a; // will be (de)serialized
  [
  int b; // transient field
  char s[256]; // transient field
  ]
  extern float d; // transient field
  char *t; // will be (de)serialized
  transientInt *n; // transient field
  [
  virtual void method(char buf[1024]); // does not create a char[1024] (de)serializer
  ]
};
```

In this example, **class** ns__myClass has three transient fields: b, s, and n which will not be (de)serialized in SOAP. Field n is transient because the type is declared within a transient block. Pointers, references, and arrays of transient types are transient. The single class method is encapsulated within [and] to prevent gSOAP from creating (de)serializers for the **char**[1024] type. gSOAP will generate (de)serializers for all types that are not declared within a [and] transient block.

13.6 Volatile Data Types

While transient data types are supposed to be hidden from gSOAP, volatile data types are visible to gSOAP but their declaration and implementation is assumed to be hidden. That is, volatile data types are assumed to be part of an existing non-modifyable software package, such as a built-in library. It would not make sense to redefine the data types in a gSOAP header file. When you need to (de)serialize such data types, you must declare them in a gSOAP header file and use the **volatile** qualifier.

Consider for example **struct** tm, declared in time.h. The structure may actually vary between platforms, but the tm structure includes at least the following fields:

```
volatile struct tm
{
  int tm_sec; /* seconds (0 - 60) */
  int tm_min; /* minutes (0 - 59) */
  int tm_hour; /* hours (0 - 23) */
  int tm_mday; /* day of month (1 - 31) */
  int tm_mon; /* month of year (0 - 11) */
  int tm_year; /* year - 1900 */
  int tm_wday; /* day of week (Sunday = 0) */
  int tm_yday; /* day of year (0 - 365) */
  int tm_isdst; /* is summer time in effect? */
  char *tm_zone; /* abbreviation of timezone name */
  long tm_gmtoff; /* offset from UTC in seconds */
};
```

Note the we qualified the structure **volatile** in the gSOAP header file to inform the gSOAP compiler that it should not attempt to redeclare it. We can now readily serialize and deserialize the tm structure. The following program fragment serializes the local time stored in a tm structure to stdout:

```
time_t T = time(NULL);
struct tm *t = localtime(&T);
struct soap *soap = soap_new();
soap_set_omode(soap, SOAP_XML_GRAPH); // good habit to use this
soap_begin_send(soap);
soap_put_tm(soap, t, "myLocalTime", NULL);
soap_end_send(soap);
soap_end(soap);
soap_done(soap);
free(soap);
```

It is also possible to serialize the tm fields as XML attributes using the @ qualifier, see Section 9.5.7.

If you must produce a schema file, say time.xsd, that defines an XML schema and namespace for the tm struct, you can add a **typedef** declaration to the header file:

```
typedef struct tm time__struct_tm;
```

We used the **typedef** name time__struct_tm rather than time__tm, because a schema name clash will occur with the latter since taking off the time prefix will result in the same name being used.

Classes should be declared volatile to prevent modification of these classes by gSOAP. gSOAP adds serialization methods to classes to support polymorphism. However, this is a problem when you can't modify class declarations because they are part of a non-modifyable software package. The solution is to declare these classes **volatile**, similar to the tm structure example illustrated above. You can also use a **typedef** to associate a schema with a class.

13.7 How to Declare User-Defined Serializers and Deserializers

Users can declare their own (de)serializers for specific data types instead of relying on the gSOAP-generated (de)serializers. To declare a external (de)serializer, declare a type with **extern typedef**. gSOAP will not generate the (de)serialzers for the type name that is declared. For example:

```
extern typedef char *MyData;
struct Sample
{
  MyData s; // use user-defined (de)serializer for this field
  char *t; // use gSOAP (de)serializer for this field
};
```

The user is required to supply the following routines for each **extern typedef**'ed name T:

```
void soap_mark_T(struct soap *soap, const T *a)
void soap_default_T(struct soap *soap, T *a)
void soap_out_T(struct soap *soap, const char *tag, int id, const T *a, const char *type)
T *soap_in_T(struct soap *soap, const char *tag, T *a, const char *type)
```

The function prototypes can be found in soapH.h.

For example, the (de)serialization of MyData can be done with the following code:

```
void soap_mark_MyData(struct soap *soap, MyData *const*a)
{ } // no need to mark this node (for multi-ref and cycle detection)
void soap_default_MyData(&soap, MyData **a)
{ *a = NULL }
void soap_out_MyData(struct soap *soap, const char *tag, int id, MyData *const*a, const char
*type)
{
  soap_element_begin_out(soap, tag, id, type); // print XML beginning tag
  soap_send(soap, *a); // just print the string (no XML conversion)
  soap_element_end_out(soap, tag); // print XML ending tag
}
MyData **soap_in_MyData(struct soap *soap, const char *tag, MyData **a, const char *type)
{
```

```
if (soap_element_begin_in(soap, tag))
  return NULL;
if (!a)
  a = (MyData**)soap_malloc(soap, sizeof(MyData*));
if (soap->null)
  *a = NULL; // xsi:nil element
if (*soap->type && soap_match_tag(soap, soap->type, type))
{
  soap->error = SOAP_TYPE_MISMATCH;
  return NULL; // type mismatch
}
if (*soap->href)
  a = (MyData**)soap_id_forward(soap, soap-¿href, a, SOAP_MyData, sizeof(MyData*))
else if (soap->body)
{
  char *s = soap_value(soap); // fill buffer
  *a = (char*)soap_malloc(soap, strlen(s)+1);
  strcpy(*a, s);
}
if (soap-¿body && soap_element_end_in(soap, tag))
  return NULL;
return a;
```

More information on custom (de)serialization will be provided in this document or in a separate document in the future. The writing of the (de)serializer code requires the use of the low-level gSOAP API.

13.8 How to Serialize Data Without Generating XSD Type Attributes

gSOAP serializes data in XML with `xsi:type` attributes when the types are declared with namespace prefixes to indicate the schema type of the data contained in the elements. SOAP 1.1 and 1.2 requires `xsi:type` attributes in the presence of polymorphic data or when the type of the data cannot be deduced from the SOAP payload. The namespace prefixes are associated with the type names of **typedef**s (Section 9.2) for primitive data types, **struct**/class names, and **enum** names.

To prevent the output of these `xsi:type` attributes in the XML serialization, you can simply use type declarations that do not include these namespace prefixes. That is, don't use the **typedef**s for primitive types and use unqualified type names with **struct**s, **class**es, and enums.

However, there are two issues. Firstly, if you want to use a primitive schema type that has no C/C++ counterpart, you must declare it as a **typedef** name with a leading underscore, as in:

```
typedef char *_xsd__date;
```

This will produce the necessary `xsd:date` information in the WSDL output by the gSOAP compiler. But the XML serialization of this type at run time won't include the `xsi:type` attribute. Secondly, to include the proper schema definitions in the WSDL produced by the gSOAP compiler, you should use qualified **struct**, **class**, and **enum** names with a leading underscore, as in:

```
struct _ns__myStruct
{ ... };
```

This ensures that myStruct is assocated with a schema, and therefore included in the appropriate schema in the generated WSDL. The leading underscore prevents the XML serialization of `xsi:type` attributes for this type in the SOAP/XML payload.

13.9 Function Callbacks for Customized I/O and HTTP Handling

gSOAP provides five callback functions for customized I/O and HTTP handling:

Callback (function pointer)
int (*soap.fopen)(**struct** soap *soap, **const char** *endpoint, **const char** *host, **int** port) Called from a client proxy to open a connection to a Web Service located at endpoint. Input parameters host and port are micro-parsed from endpoint. Should return a valid file descriptor, or -1 and soap->error set to an error code. Built-in gSOAP function: tcp_connect
int (*soap.fpost)(**struct** soap *soap, **const char** *endpoint, **const char** *host, **int** port, **const char** *path, **const char** *action, size_t count) Called from a client proxy to generate the HTTP header to connect to endpoint. Input parameters host, port, and path are micro-parsed from endpoint, action is the SOAP action, and count is the length of the SOAP message or 0 when SOAP_ENC_XML is set or when SOAP_IO_LENGTH is reset. Use function soap_send(**struct** soap *soap, **char** *s) to write the header contents. Should return SOAP_OK, or a gSOAP error code. Built-in gSOAP function: http_post.
int (*soap.fposthdr)(**struct** soap *soap, **const char** *key, **const char** *val) Called by http_post and http_response (through the callbacks). Emits HTTP key: val header entries. Should return SOAP_OK, or a gSOAP error code. Built-in gSOAP function: http_post_header.
int (*soap.fresponse)(**struct** soap *soap, **int** soap_error_code, size_t count) Called from a service to generate the response HTTP header. Input parameter soap_error_code is a gSOAP error code (see Section 8.2 and count is the length of the SOAP message or 0 when SOAP_ENC_XML is set or when SOAP_IO_LENGTH is reset. Use function soap_send(**struct** soap *soap, **char** *s) to write the header contents. Should return SOAP_OK, or a gSOAP error code Built-in gSOAP function: http_response
int (*soap.fparse)(**struct** soap *soap) Called by client proxy and service to parse an HTTP header (if present). When user-defined, this routine must at least skip the header. Use function **int** soap_getline(**struct** soap *soap, **char** *buf, **int** len) to read HTTP header lines into a buffer buf of length len (returns empty line at end of HTTP header). Should return SOAP_OK, or a gSOAP error code. Built-in gSOAP function: http_parse
int (*soap.fparsehdr)(**struct** soap *soap, **const char** *key, **const char** *val) Called by http_parse (through the fparse callback). Handles HTTP key: val header entries to set gSOAP's internals. Should return SOAP_OK, or a gSOAP error code. Built-in gSOAP function: http_parse_header
int (*soap.fclose)(**struct** soap *soap) Called by client proxy **multiple times**, to close a socket connection before a new socket connection is established and at the end of communications when the SOAP_IO_KEEPALIVE flag is not set and soap.keep_alive≠0 (indicating that the other party supports keep alive). Should return SOAP_OK, or a gSOAP error code. Built-in gSOAP function: tcp_disconnect
int (*soap.fsend)(**struct** soap *soap, **const char** *s, size_t n) Called for all send operations to emit contents of s of length n. Should return SOAP_OK, or a gSOAP error code. Built-in gSOAP function: fsend
size_t (*soap.frecv)(**struct** soap *soap, **char** *s, size_t n) Called for all receive operations to fill buffer s of maximum length n. Should return the number of bytes read or 0 in case of an error, e.g. EOF. Built-in gSOAP function: frecv
int (*soap_fignore)(**struct** soap *soap, **const char** *tag) Called when an unknown XML element was encountered on the input and tag is the offending XML element tag name. Should return SOAP_OK, or a gSOAP error code such as SOAP_TAG_MISMATCH to throw an exception. Built-in gSOAP function: none.
int (*soap_faccept)(**struct** soap *soap, **struct** sockaddr *a, **int** *n) Called by soap_accept. This is a wrapper routine for accept. Should return a valid socket descriptor or -1 and set soap->error to an error code. Built-in gSOAP function: tcp_accept

In addition, a **void***user field in the **struct** soap data structure is available to pass user-defined data to the callbacks.

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

```
char buf[10000]; // XML buffer
int len1 = 0; // #chars written
int len2 = 0; // #chars read
// mysend: put XML in buf[]
int mysend(struct soap *soap, const char *s, size_t n)
{
  if (len1 + n > sizeof(buf))
    return SOAP_EOF;
  strcpy(buf + len1, s);
  len1 += n;
  return SOAP_OK;
}
// myrecv: get XML from buf[]
size_t myrecv(struct soap *soap, char *s, size_t n)
{
  if (len2 + n > len1)
    n = len1 - len2;
  strncpy(s, buf + len2, n);
  len2 += n;
  return n;
}
main()
{
  struct soap soap;
  struct ns__person p;
  soap_init(&soap);
  len1 = len2 = 0; // reset buffer pointers
  p.name = "John Doe";
  p.age = 25;
  soap.fsend = mysend; // assign callback
  soap.frecv = myrecv; // assign callback
  soap_begin(&soap);
  soap_set_omode(&soap, SOAP_XML_GRAPH);
  soap_serialize_ns__person(&soap, &p);
  soap_put_ns__person(&soap, &p, "ns:person", NULL);
  if (soap.error)
  {
    soap_print_fault(&soap, stdout);
    exit(1);
  }
  soap_end(&soap);
  soap_begin(&soap);
  soap_get_ns__person(&soap, &p, "ns:person", NULL);
  if (soap.error)
  {
    soap_print_fault(&soap, stdout);
    exit(1);
  }
  soap_end(&soap);
  soap_init(&soap); // disable callbacks
```

```
}
```

The soap_done function can be called to reset the callback to the default internal gSOAP I/O and HTTP handlers.

The following example illustrates customized I/O and (HTTP) header handling. The SOAP request is saved to a file. The client proxy then reads the file contents as the service response. To perform this trick, the service response has exactly the same structure as the request. This is declared by the **struct** ns__test output parameter part of the remote method declaration. This struct resembles the service request (see the generated soapStub.h file created from the header file).

The header file is:

```
//gsoap ns service name: callback
//gsoap ns service namespace: urn:callback
struct ns__person
{
  char *name;
  int age;
};
int ns__test(struct ns__person in, struct ns__test &out);
```

The client program is:

```
#include "soapH.h"
...
int myopen(struct soap *soap, 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|O_CREAT, S_IWUSR|S_IRUSR)) < 0)
    return SOAP_EOF;
  return SOAP_OK;
}
void myclose(struct soap *soap)
{
  if (soap->sendfd > 2) // still open?
    close(soap->sendfd); // then close it
  soap->recvfd = 0; // set back to stdin
  soap->sendfd = 1; // set back to stdout
}
int mypost(struct soap *soap, const char *endpoint, const char *host, const char *path, const
char *action, size_t count)
{
  return soap_send(soap, "Custom-generated file\n"); // writes to soap->sendfd
}
int myparse(struct soap *soap)
{
  char buf[256];
  if (lseek(soap->recvfd, 0, SEEK_SET) < 0 —— soap_getline(soap, buf, 256)) // go to begin and
```

```
skip custom header
    return SOAP_EOF;
  return SOAP_OK;
}
main()
{
  structsoap soap;
  struct ns__test r;
  struct ns__person p;
  soap_init(&soap); // reset
  p.name = "John Doe";
  p.age = 99;
  soap.fopen = myopen; // use custom open
  soap.fpost = mypost; // use custom post
  soap.fparse = myparse; // use custom response parser
  soap.fclose = myclose; // use custom close
  soap_call_ns__test(&soap, "file://test.xml", "", p, r);
  if (soap.error)
  {
    soap_print_fault(&soap, stdout);
    exit(1);
  }
  soap_end(&soap);
  soap_init(&soap); // reset to default callbacks
}
```

SOAP 1.1 and 1.2 specify that XML elements may be ignored when present in a SOAP payload on the receiving side. gSOAP ignores XML elements that are unknown, unless the XML attribute `mustUnderstand="true"` is present in the XML element. It may be undesirable for elements to be ignored when the outcome of the omission is uncertain. The soap.fignore callback can be set to a function that returns SOAP_OK in case the element can be safely ignored, or SOAP_MUSTUNDERSTAND to throw an exception, or to perform some application-specific action. For example, to throw an exception as soon as an unknown element is encountered on the input, use:

```
int myignore(struct soap *soap, const char *tag)
{
  return SOAP_MUSTUNDERSTAND; // never skip elements (secure)
}
...
soap.fignore = myignore;
soap_call_ns__method(&soap, ...); // or soap_serve(&soap);
```

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

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

```
soap.fignore = myignore;
soap_call_ns__method(&soap, ...); // or soap_serve(&soap)
...
struct Namespace namespaces[] =
{
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
  {"SOAP-ENC","http://schemas.xmlsoap.org/soap/encoding/"},
  {"xsi", "http://www.w3.org/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 third parameter may be a pattern where * is a wildcard and - is a single character wildcard. So for example soap_match_tag(tag, "ns:*") will match any element in namespace ns or when no namespace prefix is present in the XML message.

The callback can also be used to keep track of unknown elements in an internal data structure such as a list:

```
struct Unknown
{
  char *tag;
  struct Unknown *next;
};
int myignore(struct soap *soap, const char *tag)
{
  char *s = (char*)soap_malloc(soap, strlen(tag)+1);
  struct Unknown *u = (struct Unknown*)soap_malloc(soap, sizeof(struct Unknown));
  if (s && u)
  {
    strcpy(s, tag);
    u->tag = s;
    u->next = ulist;
    ulist = u;
  }
}
...
struct soap *soap;
struct Unknown *ulist = NULL;
soap_init(&soap);
soap.fignore = myignore;
soap_call_ns__method(&soap, ...); // or soap_serve(&soap)
// print the list of unknown elements
soap_end(&soap); // clean up
```

13.10 Speed Improvement Tips

Here are some tips you can use to speed up gSOAP. gSOAP's default settings are choosen to maximize portability and compatibility. The settings can be tweeked to optimize the performance as follows:

- Increase the buffer size SOAP_BUFLEN by changing the SOAP_BUFLEN macro in stdsoap2.h. Use buffer size 65536 for example.
- Use HTTP keep-alive at the client-side, see 13.12, when the client needs to make a series of calls to the same server. Server-side keep-alive support can greatly improve performance of both client and server. But be aware that clients and services under Unix/Linux require signal handlers to catch dropped connections.
- Use HTTP chunked transfers, see 13.13.
- Do NOT use gzip compression, even when transferring data over a modem connection. Modems already compress data transfers.

13.11 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 attribute:

```
struct soap soap;
soap_init(&soap);
...
soap.http_version = "1.1";
```

13.12 HTTP Keep-Alive

gSOAP supports keep-alive socket connections. To activate keep-alive support, set the SOAP_IO_KEEPALIVE flag for both input and output modes, see Section 7.12. For example

```
struct soap soap;
soap_init2(&soap, SOAP_IO_KEEPALIVE, SOAP_IO_KEEPALIVE);
```

When a client or a service communicates with another client or service that supports keep alive, the attribute soap.keep_alive will be set to 1, otherwize it is reset to 0 (indicating that the other party will close the connection). The connection maybe terminated on either end before the communication completed, for example when the server keep-alive connection has timed out. This generates a "Broken Pipe" signal on Unix/Linux platforms. This signal can be caught with a signal handler:

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

where, for example:

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

Alternatively, broken pipes can be kept silent by setting:

```
soap.socket_flags = MSG_NOSIGNAL;
```

This setting will not generate a sigpipe but read/write operations return SOAP_EOF instead. Note that Win32 systems do not support signals and lack the MSG_NOSIGNAL flag. The sigpipe handling and flags are not very portable.

A connection will be kept open only if the request contains an HTTP 1.0 header with "`Connection: Keep-Alive`" or an HTTP 1.1 header that does not contain "`Connection: close`". This means that a gSOAP client method call should use "`http://`" in the endpoint URL of the request to the stand-alone service to ensure HTTP headers are used.

If the client does not close the connection, the server will wait forever when no recv_timeout is specified. In addition, other clients will be denied service as long as a client keeps the connection to the server open. To prevent this from happening, the service should be multi-threaded such that each thread handles the client connection:

```
int main(int argc, char **argv)
{
  struct soap soap, *tsoap;
  pthread_t tid;
  int m, s;
  soap_init2(&soap, SOAP_IO_KEEPALIVE, SOAP_IO_KEEPALIVE);
  soap.accept_timeout = 600; // optional: let server time out after ten minutes of inactivity
  m = soap_bind(&soap, NULL, 18000, BACKLOG); // use port 18000 on the current machine
  if (m < 0)
  {
    soap_print_fault(&soap, stderr);
    exit(1);
  }
  fprintf(stderr, "Socket connection successful %d\n", m);
  for (count = 0; count >= 0; count++)
  {
    soap.socket_flags = MSG_NOSIGNAL; // use this
    soap.accept_flags = SO_NOSIGPIPE; // or this to prevent sigpipe
    s = soap_accept(&soap);
    if (s < 0)
    {
      if (soap.errnum)
        soap_print_fault(&soap, stderr);
      else
        fprintf(stderr, "Server timed out\n"); // Assume timeout is long enough for threads to
complete serving requests
      break;
    }
    fprintf(stderr, "Accepts socket %d connection from IP %d.%d.%d.%d\n", s, (int)(soap.ip>>24)&0xFF,
(int)(soap.ip>>16)&0xFF, (int)(soap.ip>>8)&0xFF, (int)soap.ip&0xFF);
    tsoap = soap_copy(&soap);
    pthread_create(&tid, NULL, (void*(*)(void*))process_request, (void*)tsoap);
  }
  return 0;
}
void *process_request(void *soap)
{
  pthread_detach(pthread_self());
  ((struct soap*)soap)-¿recv_timeout = 300; // Timeout after 5 minutes stall on recv
```

```
    ((struct soap*)soap)-¿send_timeout = 60; // Timeout after 1 minute stall on send
    soap_serve((struct soap*)soap);
    soap_destroy((struct soap*)soap);
    soap_end((struct soap*)soap);
    soap_done((struct soap*)soap);
    free(soap);
    return NULL;
}
```

To prevent a malicious client from keeping a thread waiting forever by keeping the connection open, timeouts are set in the process_request routine. See Section 13.17 for more details on timeout settings.

A gSOAP client call will automatically attempt to re-establish a connection to a server when the server has terminated the connection for any reason. This way, a sequence of calls can be made to the server while keeping the connection open. Client stubs will poll the server to check if the connection is still open. When the connection was terminated by the server, the client will automatically reconnect.

A client should reset SOAP_IO_KEEPALIVE just before the last call to a server to close the connection after this last call. This will close the socket after the call and also informs the server to gracefully close the connection.

13.13 HTTP Chunked Transfer Encoding

gSOAP supports HTTP chunked transfer encoding. Un-chunking of inbound messages takes place automatically. Outbound messages are never chunked, except when the SOAP_IO_CHUNK flag is set for the output mode. Most Web services, however, will not accept chunked inbound messages.

13.14 HTTP Buffered Sends

The entire outbound message can be stored to determine the HTTP content length rather than the two-phase encoding used by gSOAP which requires a separate pass over the data to determine the length of the outbound message. Setting the flag SOAP_IO_STORE for the output mode will buffer the entire message. This can speed up the transmission of messages, depending on the content, but may require significant storage space to hold the verbose XML message.

Zlib compressed transfers require buffering. The SOAP_IO_STORE flag is set when the SOAP_ENC_ZLIB flag is set to send compressed messages. The use of chunking significantly reduces memory usage and may speed up the transmission of compressed SOAP/XML messages. This is accomplished by setting the SOAP_IO_CHUNK flag with SOAP_ENC_ZLIB for the output mode.

13.15 HTTP Authentication

HTTP authentication (basic) is enabled at the client-side by setting the soap.userid and soap.passwd strings to a username and password, respectively. A server may request user authentication and denies access (HTTP 401 error) when the client tries to connect without HTTP authentication (or with the wrong authentication information).

Here is an example client code fragment to set the HTTP authentication username and password:

```
struct soap soap;
soap_init(&soap);
soap.userid = "guest";
soap.passwd = "visit";
...
```

A client SOAP request will have the following HTTP header:

```
POST /XXX HTTP/1.0
Host:  YYY
User-Agent:  gSOAP/2.2
Content-Type:  text/xml; charset=utf-8
Content-Length:  nnn
Authorization:  Basic Z3Vlc3Q6Z3Vlc3Q=
...
```

A client MUST set the soap.userid and soap.passwd strings for each call that requires client authentication. The strings are reset after each successful or unsuccessful call.

A stand-alone gSOAP Web Service can enforce HTTP authentication upon clients, by checking the soap.userid and soap.passwd strings. These strings are set when a client request contains HTTP authentication headers. The strings SHOULD be checked in each service method (that requires authentication to execute).

Here is an example service method implementation that enforced client authentication:

```
int ns__method(struct soap *soap, ...)
{
  if (!soap->.userid —— !soap->.passwd —— strcmp(soap->.userid, "guest") —— strcmp(soap->.passwd,
"visit"))       return 401; ...
}
```

When the authentication fails, the service response with a SOAP Fault message and a HTTP error code "401 Unauthorized". The HTTP error codes are described in Section 8.2.

13.16 HTTP Proxy Authentication

HTTP proxy authentication (basic) is enabled at the client-side by setting the soap.proxy_userid and soap.proxy_passwd strings to a username and password, respectively. For example, a proxy server may request user authentication. Otherwise, access is denied by the proxy (HTTP 407 error). Example client code fragment to set proxy server, username, and password:

```
struct soap soap;
soap_init(&soap);
soap.proxy_host = "xx.xx.xx.xx"; // IP
soap.proxy_port = 8080;
soap.proxy_userid = "guest";
soap.proxy_passwd = "guest";
...
```

A client SOAP request will have the following HTTP header:

```
POST /XXX HTTP/1.0
Host:  YYY
User-Agent:  gSOAP/2.2
Content-Type:  text/xml; charset=utf-8
Content-Length:  nnn
Proxy-Authorization:  Basic Z3Vlc3Q6Z3Vlc3Q=
...
```

13.17 Timeout Management for Non-Blocking Operations

Socket connect, accept, send, and receive timeout values can be set to manage socket communication timeouts. The soap.connect_timeout, soap.accept_timeout, soap.send_timeout, and soap.recv_timeout attributes of the current gSOAP runtime environment soap can be set to the appropriate user-defined socket send, receive, and accept timeout values. A positive value measures the timeout in seconds. A negative timeout value measures the timeout in microseconds (10^{-6} sec).

The soap.connect_timeout specifies the timeout value for soap_call_ns_ _method calls.

The soap.accept_timeout specifies the timeout value for soap_accept(&soap) calls.

The soap.send_timeout and soap.recv_timeout specify the timeout values for non-blocking socket I/O operations.

Example:

```
struct soap soap;
soap_init(&soap);
soap.send_timeout = 10;
soap.recv_timeout = 10;
```

This will result in a timeout if no data can be send in 10 seconds and no data is received within 10 seconds after initiating a send or receive operation over the socket. A value of zero disables timeout, for example:

```
soap.send_timeout = 0;
soap.recv_timeout = 0;
```

When a timeout occurs in send/receive operations, a SOAP_EOF exception will be raised ("end of file or no input"). Negative timeout values measure timeouts in microseconds, for example:

```
#define uSec *-1
#define mSec *-1000
soap.accept_timeout = 10 uSec;
soap.send_timeout = 20 mSec;
soap.recv_timeout = 20 mSec;
```

The macros improve readability.

Caution: Many Linux versions do not support non-blocking connect(). Therefore, setting soap.connect_timeout for non-blocking soap_call_ns_ _method calls may not work under Linux.

13.18 Socket Options and Flags

gSOAP's socket communications can be controlled with socket options and flags. The gSOAP run-time environment **struct** soap flags are: **int** soap.socket_flags to control socket send() and recv() calls, **int** soap.connect_flags to set client connection socket options, **int** soap.bind_flags to set server-side port bind socket options, **int** soap.accept_flags to set server-side request message accept socket options. See the manual pages of send and recv for soap.socket_flags values and see the manual pages of setsockopt for soap.connect_flags, soap.bind_flags, and soap.accept_flags (SOL_SOCKET) values. These SO_ socket option flags (see setsockopt manual pages) can be bit-wise or-ed to set multiple socket options at once. The client-side flag soap.connect_flags=SO_LINGER is supported with values l_onoff=1 and l_linger=0.

For example, to disable sigpipe signals on Unix/Linux platforms use: soap.socket_flags=MSG_NOSIGNAL and/or soap.connect_flags=SO_NOSIGPIPE (i.e. client-side connect) depending on your platform.

Use soap.bind_flags=SO_REUSEADDR to enable server-side port reuse and local port sharing (but be aware of the security issues when the port is not blocked by a firewall and open to the Internet).

13.19 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 all the sources of 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 soapdefs.h:

```
#define WITH_OPENSSL
```

and compile with option -DWITH_SOAPDEFS_H to include soapdefs.h in your project. A client program simply uses the prefix https: instead of http: in the endpoint URL of a remote method call to a Web Service to use encrypted transfers (if the service supports HTTPS). For example:

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

By default, server authentication is disabled. To enable server authentication, set the require_server_auth attribute of the current gSOAP runtime environment (**struct** soap) before a call is made:

```
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) { }
```

Caution: it is important that the WITH_OPENSSL macro MUST be consistently defined to compile the sources, such as stdsoap2.cpp, soapC.cpp, soapClient.cpp, soapServer.cpp, and all application sources that include stdsoap2.h or soapH.h. If the macros are not consistently used, the application will crash due to a mismatches in the declaration and access of the gSOAP environment. **Caution**: concurrent client calls MUST be made using separate soap structs (and they should not be copied using soap_copy from an existing soap struct).

13.20 Secure SOAP Web Services with HTTPS/SSL

When a Web Service is installed as CGI, it uses standard I/O that is encryped/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 (if RSA is not used), and password need to be supplied. Instructions on how to do this can be found in the OpenSSL documentation http://www.openssl.org. See also Section 13.22. To enable OpenSSL, first install OpenSSL and use option -DWITH_OPENSSL to compile the sources with your C or C++ compiler, for example:

```
g++ -DWITH_OPENSSL -o myprog myprog.cpp stdsoap2.cpp soapC.cpp soapServer.cpp -lssl -
lcrypto
```

Let's take a look at an example SSL secure multi-threaded stand-alone SOAP Web Service:

```
int main()
{
  int m, s;
  pthread_t tid;
  struct soap soap, *tsoap;
  CRYPTO_thread_setup(); // see text
  soap_init(&soap);
  // soap.rsa = 1; // use RSA (or use DH which requires a DH file: see below)
  soap.keyfile = "server.pem"; // must be resident key file
  soap.cafile = "cacert.pem"; // must be resident CA file
  soap.dhfile = "dh512.pem"; // if soap.rsa == 0, use DH with resident DH file
  soap.password = "password"; // password
  soap.randfile = "random.rnd"; // (optional) some file with random data to seed PRNG
  m = soap_bind(&soap, NULL, 18000, 100); // use port 18000
  if (m < 0)
  {
    soap_print_fault(&soap, stderr);
    exit(1);
  }
  fprintf(stderr, "Socket connection successful: master socket = %d\n", m);
  for (;;)
```

```
  {
    s = soap_accept(&soap);
    fprintf(stderr, "Socket connection successful: slave socket = %d\n", s);
    if (s < 0)
    {
      soap_print_fault(&soap, stderr);
      break;
    }
    tsoap = soap_copy(&soap); // should call soap_ssl_accept on a copy
    if (!tsoap)
      break;
    if (soap_ssl_accept(tsoap))
    {
      soap_print_fault(tsoap, stderr);
      soap_done(tsoap);
      free(tsoap);
      continue; // when soap_ssl_accept fails, we should just go on
    }
    pthread_create(&tid, NULL, &process_request, (void*)tsoap);
  }
  CRYPTO_thread_cleanup(); // see text
  soap_done(&soap); // deallocates SSL
  return 0;
}
void *process_request(void *soap)
{
  pthread_detach(pthread_self());
  soap_serve((struct soap*)soap);
  soap_destroy((struct soap*)soap);
  soap_end((struct soap*)soap);
  soap_done((struct soap*)soap);
  free(soap);
  return NULL;
}
```

The CRYPTO_thread_setup() and CRYPTO_thread_cleanup() routines can be found in openssl/crypto/threads/th-lock.c. These routines are required to setup locks for multi-threaded applications that use SSL. We give a Pthreads-based implementation of these here:

```
static pthread_mutex_t *lock_cs;
static long *lock_count;
void locking_function(int, int, const char*, int);
unsigned long id_function();
void CRYPTO_thread_setup()
{
  int i;
  lock_cs = (pthread_mutex_t*)OPENSSL_malloc(CRYPTO_num_locks() * sizeof(pthread_mutex_t));
  lock_count = (long*)OPENSSL_malloc(CRYPTO_num_locks() * sizeof(long));
  for (i = 0; i < CRYPTO_num_locks(); i++)
  {
    lock_count[i] = 0;
    pthread_mutex_init(&(lock_cs[i]), NULL);
  }
```

```
  CRYPTO_set_id_callback(id_function);
  CRYPTO_set_locking_callback(locking_function);
}
void CRYPTO_thread_cleanup()
{
  int i;
  CRYPTO_set_locking_callback(NULL);
  for (i = 0; i < CRYPTO_num_locks(); i++)
    pthread_mutex_destroy(&(lock_cs[i]));
  OPENSSL_free(lock_cs);
  OPENSSL_free(lock_count);
}
void locking_function(int mode, int type, const char *file, int line)
{
  if (mode & CRYPTO_LOCK)
  {
    pthread_mutex_lock(&(lock_cs[type]));
    lock_count[type]++;
  }
  else
    pthread_mutex_unlock(&(lock_cs[type]));
}
unsigned long id_function()
{ return (unsigned long)pthread_self();
}
```

When Web services need to verify clients you can use a key file, CA file, a file with random data, and password in an SSL-enabled client:

```
...
soap_init(&soap);
soap.keyfile = "client.pem";
soap.password = "password";
soap.cafile = "cacert.pem";
soap.randfile = "random.rnd";
if (soap_call_ns__method(&soap, "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):

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

where, for example:

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

Caution: it is important that the WITH_OPENSSL macro MUST be consistently defined to compile the sources, such as stdsoap2.cpp, soapC.cpp, soapClient.cpp, soapServer.cpp, and all application sources that include stdsoap2.h or soapH.h. If the macros are not consistently used, the application will crash due to a mismatches in the declaration and access of the gSOAP environment.

13.21 SSL Authentication Callback

gSOAP provides a callback function for authentication initialization:

Callback (function pointer)
int (*soap.fsslauth)(**struct** soap *soap) Initialize the authentication information for clients and services, such as the certificate chain, password, read the key and/or DH file, generate an RSA key, and initialization of the RNG. Should return a gSOAP error code or SOAP_OK. Built-in gSOAP function: ssl_auth_init

13.22 SSL Certificates

The .pem files in the gSOAP distribution are examples. Developers will have to generate certificates as needed (client only, server only or both). There is more than one way to generate pem files for clients and servers. Here is the simplest/quickest one:

Create a private Certificate Authority (CA). The CA is used in SSL to verify the authenticity of a given certificate. The CA acts as a trusted third party who has authenticated the user of the signed certificate as being who they say. The certificate is signed by the CA, and if the client trusts the CA, it will trust your certificate. For use within your organization, a private CA will probably serve your needs. However, if you intend use your certificates for a public service, you should probably obtain a certificate from a known CA (e.g. VeriSign). In addition to identification, your certificate is also used for encryption.

Creating a private CA:

- Go to the OpenSSL bin directory (/usr/local/ssl/misc by default and /System/Library/OpenSSL/misc on Mac OS X).
- There should be a script called CA.sh (and a CA.pl that does the same stuff). This hides all the gruesome details of how this works. Without the script this is a very annoying process.
- su to root
- Make sure that the OpenSSL bin directory is in your path.
- ./CA.sh -newca
- When prompted for CA filename hit return.
- Answer the rest of the questions intelligently. The common name would be how this certificate might be referred to. For example, the Equifax Secure CA uses the common name of Equifax Secure Certificate Authority. Do not forget the password!

Creating certificates should be done through a certificate authority to obtain signed certificates. But you can create your own certificates for testing purposes as follows.

- ./CA.sh -newreq
- This creates an unsigned certificate request.

- The procedure is the same as creating a private CA except you'll want to use the name of the host that will use the certificate as the common name. If they don't match, the client will not like it when you set require_server_auth=1.
- You probably don't want to use the same passphrase for this as you did with the CA.
- ./CA.sh -sign
- It will ask for a PEM pass phrase, that's the passphrase you set for the private CA you created.
- This signs the certificate that you just created with the CA you created just moments before.
- The signed certificate is now in the current directory as newcert.pem. If you are going to create more, you should rename this or it will be overwritten be subsequent signatures.

Now do the following at the prompt:

cat newreq.pem newcert.pem > server.pem (or client.pem as needed)

You now have generated a client and a server certificate in PEM format by repeating the same process and changing the common name for the client (e.g. gSOAP client). You also need the CA certificate.

Finally you need to generate Diffie-Helmann parameters for the server. Do the following at the prompt:

openssl dhparam -outform PEM -out dh.pem 1024

File dh.pem is the output file and 1024 is the number of bits used (this will take a long time, you can safely use 512).

Of course the developer using your server cert on her machine will find that if require_server_auth=1 the client will exit before doing the handshake.

13.23 Zlib Compressed Messages

To enable deflate and gzip compression with Zlib, install Zlib from `http://www.zlib.org` if not already installed on your system. Compile stdsoap2.cpp (or stdsoap2.c) and **all** your sources that include stdsoap2.h or soapH.h with compiler option -DWITH_GZIP and link your code with the Zlib library, e.g. -lz on Unix/Linux platforms.

The gzip compression is orthogonal to all transport encodings such as HTTP, SSL, DIME, and can be used with other transport layers. You can even save and load compressed XML data to/from files.

gSOAP supports two compression formats: deflate and gzip. The gzip format is used by default. The gzip format has several benefits over deflate. Firstly, gSOAP can automatically detect gzip compressed inbound messages, even without HTTP headers, by checking for the presence of a gzip header in the message content. Secondly, gzip includes a CRC32 checksum to ensure messages have been correctly received. Thirdly, gzip compressed content can be decompressed with other compression software, so you can decompress XML data saved by gSOAP in gzip format.

Gzip compression is enabled by compiling the sources with -DWITH_GZIP. To transmit gzip compressed SOAP/XML data, set the output mode flags to SOAP_ENC_ZLIB. For example:

```
soap_init(&soap);
...
soap_set_omode(&soap, SOAP_ENC_ZLIB); // enable Zlib's gzip
if (soap_call_ns__myMethod(&soap, ...))
...
soap_clr_omode(&soap, SOAP_ENC_ZLIB); // disable Zlib's gzip
...
```

This will send a compressed SOAP/XML request to a service, privided that Zlib is installed and linked with the application and the -DWITH_GZIP option was used to compile the sources. Receiving compressed SOAP/XML over HTTP either in gzip or deflate formats is automatic. The SOAP_ENC_ZLIB flag does not have to be set at the server side to accept compressed messages. Reading and receiving gzip compressed SOAP/XML without HTTP headers (e.g. with other transport protocols) is also automatic.

To control the level of compression for outbound messages, you can set the soap.z_level to a value between 1 and 9, where 1 is the best speed and 9 is the best compression (default is 6). For example

```
soap_init(&soap);
...
soap_set_omode(&soap, SOAP_ENC_ZLIB);
soap.z_level = 9; // best compression
...
```

To verify and monitor compression rates, you can use the values soap.z_ratio_in and soap.z_ratio_out. These two float values lie between 0.0 and 1.0 and express the ratio of the compressed message length over uncompressed message length.

```
soap_call_ns__myMethod(&soap, ...);
...
printf("Compression ratio: %f%% (in) %f%% (out)\n", 100*soap.z_ratio_out, 100*soap.z_ratio_in);
...
```

Note: lower ratios mean higher compression rates.

Compressed transfers require buffering the entire output message to determine HTTP message length. This means that the SOAP_IO_STORE flag is automatically set when the SOAP_ENC_ZLIB flag is set to send compressed messages. The use of HTTP chunking significantly reduces memory usage and may speed up the transmission of compressed SOAP/XML messages. This is accomplished by setting the SOAP_IO_CHUNK flag with SOAP_ENC_ZLIB for the output mode. However, some Web servers do not accept HTTP chunked request messages (even when they return HTTP chunked messages!). Stand-alone gSOAP services always accept chunked request messages.

To restrict the compression to the deflate format only, compile the sources with -DWITH_ZLIB. This limits compression and decompression to the deflate format. Only plain and deflated messages can be exchanged, gzip is not supported with this option. Receiving gzip compressed content is automatic, even in the absence of HTTP headers. Receiving deflate compressed content is not automatic in the absence of HTTP headers and requires the flag SOAP_ENC_ZLIB to be set for the input mode to decompress deflated data.

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

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

13.24 Client-Side Cookie Support

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

```
g++ -DWITH_COOKIES -o myclient stdsoap2.cpp soapC.cpp soapClient.cpp
```

or add the following line to stdsoap.h:

```
#define WITH_COOKIES
```

Client-side cookie support is fully automatic. So just (re)compile stdsoap2.cpp with -DWITH_COOKIES to enable cookie-based session control in your client.

A database of cookies is kept and returned to the appropriate servers. Cookies are not automatically saved to a file by a client. An example cookie file manager is included as an extras in the distribution. You should explicitly remove all cookies before terminating a gSOAP environment by calling soap_free_cookies(soap) or by calling soap_done(soap).

To avoid "cookie storms" caused by malicious servers that return an unreasonable amount of cookies, gSOAP clients/servers are restricted to a database size that the user can limit (32 cookies by default), for example:

```
struct soap soap;
soap_init(&soap);
soap.cookie_max = 10;
```

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

```
struct soap_cookie
{
  char *name;
  char *value;
  char *domain;
  char *path;
  long expire; /* client-side: local time to expire; server-side: seconds to expire */
  unsigned int version;
  short secure;
  short session; /* server-side */
  short env; /* server-side: 1 = got cookie from client */
  short modified; /* server-side: 1 = client cookie was modified */
  struct soap_cookie *next;
};
```

Since the cookie database is linked to a soap struct, each thread has a local cookie database in a multi-threaded implementation.

13.25 Server-Side Cookie Support

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

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

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

Function
struct soap_cookie *soap_set_cookie(**struct** soap *soap, **const char** *name, **const char** *value, **const char** *domain, **const char** *path); Add a cookie to the database with name name and value value. domain and path may be NULL to use the current domain and path given by soap_cookie_domain and soap_cookie_path. If successful, returns pointer to a cookie node in the linked list, or NULL otherwise.
struct soap_cookie *soap_cookie(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path); Find a cookie in the database with name name and value value. domain and path may be NULL to use the current domain and path given by soap_cookie_domain and soap_cookie_path. If successful, returns pointer to a cookie node in the linked list, or NULL otherwise.
char *soap_cookie_value(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path); Get value of a cookie in the database with name name. domain and path may be NULL to use the current domain and path given by soap_cookie_domain and soap_cookie_path. If successful, returns the string pointer to the value, or NULL otherwise.
long soap_cookie_expire(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path); Get expiration value of the cookie in the database with name name (in seconds). domain and path may be NULL to use the current domain and path given by soap_cookie_domain and soap_cookie_path. Returns the expiration value, or -1 if cookie does not exist.
int soap_set_cookie_expire(**struct** soap *soap, **const char** *name, **long** expire, **const char** *domain, **const char** *path); Set expiration value expire of the cookie in the database with name name (in seconds). domain and path may be NULL to use the current domain and path given by soap_cookie_domain and soap_cookie_path. If successful, returns SOAP_OK, or SOAP_EOF otherwise.
int soap_set_cookie_session(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path); Set cookie in the database with name name to be a session cookie. This means that the cookie will be returned to the client. (Only cookies that are modified are returned to the client). domain and path may be NULL to use the current domain and path given by soap_cookie_domain and soap_cookie_path. If successful, returns SOAP_OK, or SOAP_EOF otherwise.
int soap_clr_cookie_session(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path); Clear cookie in the database with name name to be a session cookie. domain and path may be NULL to use the current domain and path given by soap_cookie_domain and soap_cookie_path. If successful, returns SOAP_OK, or SOAP_EOF otherwise.
void soap_clr_cookie(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path); Remove cookie from the database with name name. domain and path may be NULL to use the current domain and path given by soap_cookie_domain and soap_cookie_path.
int soap_getenv_cookies(**struct** soap *soap); Initializes cookie database by reading the 'HTTP_COOKIE' environment variable. This provides a means for a CGI application to read cookies send by a client. If successful, returns SOAP_OK, or SOAP_EOF otherwise.
void soap_free_cookies(**struct** soap *soap); Release cookie database.

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

Attribute	value
const char *cookie_domain	MUST be set to the domain (host) of the service
const char *cookie_path	MAY be set to the default path to the service
int cookie_max	maximum cookie database size (default=32)

The cookie_path value is used to filter cookies intended for this service according to the path prefix rules outlined in RFC2109.

The following example server adopts cookies for session control:

```
int main()
{
  struct soap soap;
  int m, s;
  soap_init(&soap);
  soap.cookie_domain = "...";
  soap.cookie_path = "/"; // the path which is used to filter/set cookies with this destination
  if (argc < 2)
  {
    soap_getenv_cookies(&soap); // CGI app: grab cookies from 'HTTP_COOKIE' env var
    soap_serve(&soap);
  }
  else
  {
    m = soap_bind(&soap, NULL, atoi(argv[1]), 100);
    if (m < 0)
      exit(1);
    for (int i = 1; ; i++)
    {
      s = soap_accept(&soap);
      if (s < 0)
        exit(1);
      soap_serve(&soap);
      soap_end(&soap); // clean up
      soap_free_cookies(&soap); // remove all old cookies from database so no interference occurs
with the arrival of new cookies
    }
  }
  return 0;
}
int ck__demo(struct soap *soap, ...)
{
  int n;
  const char *s;
  s = soap_cookie_value(soap, "demo", NULL, NULL); // cookie returned by client?
  if (!s)
    s = "init-value"; // no: set initial cookie value
  else
    ... // modify 's' to reflect session control
  soap_set_cookie(soap, "demo", s, NULL, NULL);
  soap_set_cookie_expire(soap, "demo", 5, NULL, NULL); // cookie may expire at client-side in 5
seconds
```

```
  return SOAP_OK;
}
```

13.26 Connecting Clients Through Proxy Servers

When a client needs to connect to a Web Service through a proxy server, set the soap.proxy_host string and soap.proxy_port integer attributes of the current soap runtime environment to the proxy's host name and port, respectively. For example:

```
struct soap soap;
soap_init(&soap);
soap.proxy_host = "proxyhostname";
soap.proxy_port = 8080;
if (soap_call_ns__method(&soap, "http://host:port/path", "action", ...))
  soap_print_fault(&soap, stderr);
else
  ...
```

The attributes soap.proxy_host and soap.proxy_port keep their values throug the remove method calls, so they only need to be set once.

13.27 FastCGI Support

To enable FastCGI support, install FastCGI and compile *all* sources (including your application sources that use stdsoap2.h) with option -DWITH_FASTCGI or add

```
#define WITH_FASTCGI
```

to stdsoap2.h.

13.28 How to Create gSOAP Applications With a Small Memory Footprint

To compile gSOAP applications intended for small memory devices, you may want to remove all non-essential features that consume precious code and data space. To do this, compile the gSOAP sources with -DWITH_LEAN to remove many non-essential features. The features that will be disabled are:

- No I/O timeouts. Note that many socket operations already obey some form of timeout handling, such as a connect timeout for example.
- No HTTP keep alive
- No HTTP cookies
- No HTTP authentication
- No HTTP chunked output (but input is OK)
- No HTTP compressed output (but input is OK when compiled with WITH_GZIP)

- No socket flags (no soap.socket_flag, soap.connect_flag, soap.bind_flag, soap.accept_flag)
- No canonical XML output
- No logging
- Limited TCP/IP and HTTP error diagnostic messages
- No support for time_t serialization
- No support for LONG64 serialization
- No support for **unsigned short** and ULONG64 serialization (use **typedef unsigned int** xsd__unsignedShort or **typedef long** xsd__unsignedLong)

Use -DWITH_LEANER to make the executable even smaller by removing DIME attachment handling. Note that DIME attachments are not essential to achieve SOAP/XML interoperability. DIME attachments are a convenient way to exchange non-text-based (i.e. binary) content, but are not required for basic SOAP/XML interoperability. The DIME requirements are predictable. That is, applications won't suddenly decide to use DIME instead of XML to exchange content.

It is safe to try to compile your application with -DWITH_LEAN, provided that your application does not rely on I/O timeouts. When no linkage error occurs in the compilation process, it is safe to assume that your application will run just fine.

13.29 Build a Client or Server in a C++ Code Namespace

You can use a C++ code namespace of your choice in your header file to build a client or server in that code namespace. In this way, you can create multiple clients and servers that can be combined and linked together without conflicts, which is explained in more detail in the next section (which also shows an example combining two client libraries defined in two C++ code namespaces).

At most one namespace can be defined for the entire gSOAP header file. The code namespace MUST completely encapsulate the entire contents of the header file:

```
namespace myNamespaceName {
... gSOAP header file contents ...
}
```

When compiling this header file with the gSOAP compiler, all type definitions, the (de)serializers for these types, and the stub/skeleton codes will be placed in this namespace. The XML namespace mapping table (saved in a .nsmap file) will not be placed in the code namespace to allow it to be linked as a global object. You can use option -n to create local XML namespace tables, see Section 7.1 (but remember that you explicitly need to initialize the soap.namespaces to point to a table at run time). The generated files are prefixed with the code namespace name instead of the usual soap file name prefix to enable multiple client/server codes to be build in the same project directory (a code namespace automatically sets the -p compiler option, see Section 7.1 for options).

Because the SOAP Header and Fault serialization codes will also be placed in the namespace, they cannot be called from the stdsoap2.cpp run time library code and are therefore rendered unusable. Therefore, these serializers are not compiled at all (enforced with #define WITH_NOGLOBAL). To add

SOAP Header and Fault serializers, you MUST compile them separately as follows. First, create a new header file env.h with the SOAP Header and Fault definitions. You can leave this header file empty if you want to use the default SOAP Header and Fault. Then compile this header file with:

```
soapcpp2 -penv env.h
```

The generated envC.cpp file holds the SOAP Header and Fault serializers and you can link this file with your client or server application.

13.30 How to Create Client/Server Libraries

The gSOAP compiler produces soapClientLib.cpp and soapServerLib.cpp codes that are specifically intended for building static or dynamic client/server libraries. These codes export the stubs and skeletons, but keep all marshaling code (i.e. parameter serializers and deserializers) local (i.e. as static functions) to avoid link symbol conflicts when combining multiple clients and/or servers into one executable.

To build multiple libraries in the same project directory, you can define a C++ code namespace in your header file (see Section 13.29) or you can use soapcpp2 with option -p to rename the generated soapClientLib.cpp and soapServerLib.cpp (and associated) files. The -p option specifies the file name prefix to replace the soap prefix. The libraries don't have to be C++ codes. You can use option -c to generate C code. A clean separation of libaries can also be achieved with C++ code namespaces, see Section 13.29.

The library codes do not define SOAP Header and Fault serializers. You MUST add SOAP Header and Fault serializers to your application, which are compiled separately as follows. First, create a new header file env.h with the SOAP Header and Fault definitions. You can leave this header file empty if you want to use the default SOAP Header and Fault. Then compile this header file with:

```
soapcpp2 -penv env.h
```

The generated envC.cpp file holds the SOAP Header and Fault serializers and you can create a (dynamic) library for it to link this code with your client or server application.

You MUST compile the stdsoap2.cpp library using -DWITH_NONAMESPACES:

```
g++ -DWITH_NONAMESPACES -c stdsoap2.cpp
```

This omits the reference to the global namespaces table, which is nowhere defined. You MUST explicitly set the namespaces value of the gSOAP environment in your code every time after initialization of the soap struct:

```
soap_init(&soap);
soap.namespaces = namespaces;
```

where the namespaces[] table is defined in the client/server source, e.g. by including the generated .nsmap files. It is important to rename the tables using option -n, see Section 7.1, to avoid namespace table name clashes.

Note: Link conflicts may still occur in the unlikely situation that identical remote method names are defined in two or more client stubs or server skeletons when these methods share the same XML namespace prefix. You may have to use C++ code namespaces to avoid these link conflicts or rename the namespace prefixes used by the remote method defined in the header files.

13.30.1 C++ Example

As an example we will build a Delayed Stock Quote client library and a Currency Exchange Rate client library.

First, we create an empty header file env.h (which may contain optional SOAP Header and Fault definitions), and compile it as follows:

```
soapcpp2 -penv env.h
g++ -c envC.cpp
```

We also compile stdsoap2.cpp without namespaces:

```
g++ -c -DWITH_NONAMESPACES stdsoap2.cpp
```

Note: when you forget to use -DWITH_NONAMESPACES you will get an unresolved link error for the global namespaces table. You can define a dummy table to avoid having to recompile stdsoap2.cpp.

Second, we create the Delayed Stock Quote header file specification, which may be obtained using the WSDL importer. If you want to use C++ namespaces then you need to manually add the **namespace** declaration to the generated header file:

```
namespace quote {
//gsoap ns service name: Service
//gsoap ns service location: http://services.xmethods.net/soap
//gsoap ns schema namespace: urn:xmethods-delayed-quotes
//gsoap ns service method-action: getQuote ""
int ns__getQuote(char *symbol, float &Result);
}
```

We then compile it as a library and we use option -n to rename the namespace table to avoid link conflicts later:

```
soapcpp2 -n quote.h
g++ -c quoteClientLib.cpp
```

If you don't want to use a C++ code namespace, you should compile quote.h "as is" with soapcpp2 option -pquote:

```
soapcpp2 -n -pquote quote.h
g++ -c quoteClientLib.cpp
```

Third, we create the Currency Exchange Rate header file specification:

```
namespace rate {
//gsoap ns service name: Service
//gsoap ns service location: http://services.xmethods.net/soap
//gsoap ns schema namespace: urn:xmethods-CurrencyExchange
//gsoap ns service method-action: getRate ""
int ns__getRate(char *country1, char *country2, float &Result);
}
```

Similar to the Quote example above, we compile it as a library and we use option -n to rename the namespace table to avoid link conflicts:

```
soapcpp2 -n rate.h
g++ -c rateClientLib.cpp
```

Fourth, we consider linking the libraries to the main program. The main program can import the quoteServiceProxy.h and rateServiceProxy.h files to obtain client proxies to invoke the services. The proxy implementations are defined in the quoteClientLib.cpp and rateClientLib.cpp files compiled above. The -n option also affects the generation of the quoteServiceProxy.h and rateServiceProxy C++ proxy codes to ensure that the gSOAP environment is properly initialized with the namespace table (so you don't have to initialize explicitly – this feature is only available with C++ proxy and server object classes).

```
#include "quoteServiceProxy.h" // get quote Service proxy
#include "rateServiceProxy.h" // get rate Service proxy
#include "quote.nsmap" // get quote namespace bindings
#include "rate.nsmap" // get rate namespace bindings
int main(int argc, char *argv[])
{
  if (argc <= 1)
  {
    std::cerr << "Usage: main ticker [currency]" << std::endl
    exit(0);
  }
  quote::Service quote;
  float q;
  if (quote.getQuote(argv[1], q)) // get quote
    soap_print_fault(quote.soap, stderr);
  else
  {
    if (argc > 2)
    {
      rate::Service rate;
      float r;
      if (rate.getRate("us", argv[2], r)) // get rate in US dollars
        soap_print_fault(rate.soap, stderr);
      else
        q *= r; // convert the quote
    }
    std::cout << argv[1] << ": " << q << std::endl;
  }
  return 0;
}
```

Compile and link this application with stdsoap2.o, envC.o, quoteClientLib.o, and rateClientLib.o.

To compile and link a server object is very similar. For example, assume that we need to implement a calculator service and we want to create a library for it.

```
namespace calc {
//gsoap ns service name: Service
```

```
//gsoap ns service location: http://www.cs.fsu.edu/ engelen/calc.cgi
//gsoap ns schema namespace: urn:calc
int ns__add(double a, double b, double &result);
int ns__sub(double a, double b, double &result);
int ns__mul(double a, double b, double &result);
int ns__div(double a, double b, double &result);
}
```

We compile this with:

```
soapcpp2 -n calc.h
g++ -c calcServerLib.cpp
```

The effect of the -n option is that it creates local namespace tables, and a modified calcServiceObject.h server class that properly initializes the gSOAP run time with the table, and renames the global soap_serve function to calc_serve. Function calc_serve is invoked with the serve method of the generated calc::Service class:

```
#include "calcServiceObject.h" // get Service object
#include "calc.nsmap" // get calc namespace bindings
...
calc::Service calc;
calc.serve(); // calls request dispatcher to invoke one of the functions below
...
int calc::ns__add(struct soap *soap, double a, double b, double &result);
{ result = a + b; returnSOAP_OK; }
int calc::ns__sub(struct soap *soap, double a, double b, double &result);
{ result = a - b; returnSOAP_OK; }
int calc::ns__mul(struct soap *soap, double a, double b, double &result);
{ result = a * b; returnSOAP_OK; }
int calc::ns__div(struct soap *soap, double a, double b, double &result);
{ result = a / b; returnSOAP_OK; }
```

In fact, the calc::Service class is derived from the **struct** soap and passed to the remote method implementations. So when you invoke calc.serve() you can be assured that the calc::Service object is passed to the remote service functions.

13.30.2 C Example

This is the same example as above, but the clients are build with pure C.

First, we create an empty header file (which may contain optional SOAP Header and Fault definitions), and compile it as follows:

```
soapcpp2 -c -penv env.h
gcc -c envC.c
```

We also compile stdsoap2.c without namespaces:

```
gcc -c -DWITH_NONAMESPACES stdsoap2.c
```

Second, we create the Delayed Stock Quote header file specification, which may be obtained using the WSDL importer.

```
//gsoap ns service name: Service
//gsoap ns service location: http://services.xmethods.net/soap
//gsoap ns schema namespace: urn:xmethods-delayed-quotes
//gsoap ns service method-action: getQuote ""
int ns__getQuote(char *symbol, float *Result);
```

We compile it as a library and we use options -n and -p to rename the namespace table to avoid link conflicts:

```
soapcpp2 -c -n -pquote quote.h
gcc -c quoteClientLib.c
```

Third, we create the Currency Exchange Rate header file specification:

```
//gsoap ns service name: Service
//gsoap ns service location: http://services.xmethods.net/soap
//gsoap ns schema namespace: urn:xmethods-CurrencyExchange
//gsoap ns service method-action: getRate ""
int ns__getRate(char *country1, char *country2, float *Result);
```

We compile it as a library and we use options -n and -p to rename the namespace table to avoid link conflicts:

```
soapcpp2 -c -n -prate rate.h
gcc -c rateClientLib.c
```

The main program is:

```
#include "quoteStub.h" // get quote Service stub
#include "rateStub.h" // get rate Service stub
#include "quote.nsmap" // get quote namespace bindings
#include "rate.nsmap" // get rate namespace bindings
int main(int argc, char *argv[])
{
  if (argc <= 1)
  {
    fprintf(stderr, "Usage: main ticker [currency]\n");
    exit(0);
  }
  struct soap soap;
  float q;
  soap_init(&soap);
  soap.namespaces = quote_namespaces;
  if (soap_call_ns__getQuote(&soap, "http://services.xmethods.net/soap", "", argv[1], &q)) // get
quote
    soap_print_fault(&soap, stderr);
  else
  {
    if (argc > 2)
```

```
    {
      soap.namespaces = rate_namespaces;
      float r;
      if (soap_call_ns__getRate(&soap, "http://services.xmethods.net/soap", "", "us", argv[2],
&r)) // get rate in US dollars
        soap_print_fault(&soap, stderr);
      else
        q *= r; // convert the quote
    }
    printf("%s: %f \n", argv[1], q);
  }
  return 0;
}
```

Compile and link this application with stdsoap2.o, envC.o, quoteClientLib.o, and rateClientLib.o.

To compile and link a server library is very similar. Assuming that the server is named "calc" (as specified with options -n and -p), the application needs to include the calcStub.h file, link the calcServerLib.o file, and call calc_serve(&soap) function at run time.

13.31 How to Create DLLs

13.31.1 Create the Base stdsoap2.dll

First, create a new header file env.h with the SOAP Header and Fault definitions. You can leave this header file empty if you want to use the default SOAP Header and Fault. Then compile this header file with:

```
soapcpp2 -penv env.h
```

The generated envC.cpp file holds the SOAP Header and Fault serializers, which need to be part of the base library functions.

The next step is to create stdsoap2.dll which consists of the file stdsoap2.cpp and envC.cpp. This DLL contains all common functions needed for all other clients and servers based on gSOAP. Compile envC.cpp and stdsoap2.cpp into stdsoap2.dll using the C++ compiler option -DWITH_NONAMESPACES and the MSVC Pre-Processor definitions SOAP_FMAC1=__declspec(dllexport) and SOAP_FMAC3=__declspec(dllexport) (or you can compile with -DWITH_SOAPDEFS_H and put the macro definitions in soapdefs.h). This exports all functions which are preceded by the macro SOAP_FMAC1 in the soapcpp2.cpp source file and macro SOAP_FMAC3 in the envC.cpp source file.

13.31.2 Creating Client and Server DLLs

Compile the soapClientLib.cpp and soapServerLib.cpp sources as DLLs by using the MSVC Pre-Processor definitions SOAP_FMAC5=__declspec(dllexport) and SOAP_CMAC=__(dllexport), and by using the C++ compiler option -DWITH_NONAMESPACES. This DLL links to stdsoap2.dll.

To create multiple DLLs in the same project directory, you SHOULD use option -p to rename the generated soapClientLib.cpp and soapServerLib.cpp (and associated) files. The -p option specifies the file name prefix to replace the soap prefix. A clean separation of libaries can also be achieved with C++ namespaces, see Section 13.29.

Unless you use the client proxy and server object classes (soapXProxy.h and soapXObject.h where X is the name of the service), all client and server applications MUST explicitly set the namespaces value of the gSOAP environment:

```
soap_init(&soap);
soap.namespaces = namespaces;
```

where the namespaces[] table should be defined in the client/server source. These tables are generated in the .nsmap files. You can rename the tables using option -n, see Section 7.1.

13.31.3 gSOAP Plug-ins

The gSOAP plug-in feature enables a convenient extension mechanism of gSOAP capabilities. When the plug-in registers with gSOAP, it has full access to the run-time settings and the gSOAP function callbacks. Upon registry, the plug-in's local data is associated with the gSOAP run-time. By overriding gSOAP's function callbacks with the plug-in's function callbacks, the plug-in can extend gSOAP's capabilities. The local plug-in data can be accessed through a lookup function, usually invoked within a callback function to access the plug-in data. The registry and lookup functions are:

```
int soap_register_plugin_arg(struct soap *soap, int (*fcreate)(struct soap *soap, struct soap_plugin
*p, void *arg), void *arg)
void* soap_lookup_plugin(struct soap*, const char*);
```

Other functions that deal with plug-ins are:

```
int soap_copy(struct soap *soap);
void soap_done(struct soap *soap);
```

The soap_copy function returns a new dynamically allocated gSOAP environment that is a copy of another, such that no data is shared between the copy and the original environment. The soap_copy function invokes the plug-in copy callbacks to copy the plug-ins' local data. The soap_copy function returns a gSOAP error code or SOAP_OK. The soap_done function de-registers all plugin-ins, so this function should be called to cleanly terminate a gSOAP run-time environment.

An example will be used to illustrate these functions. This example overrides the send and receive callbacks to copy all messages that are sent and received to the terminal (stderr).

First, we write a header file plugin.h to define the local plug-in data structure(s) and we define a global name to identify the plug-in:

```
#include "stdsoap2.h"
#define PLUGIN_ID "PLUGIN-1.0" // some name to identify plugin
struct plugin_data // local plugin data
{
  int (*fsend)(struct soap*, const char*, size_t); // to save and use send callback
  size_t (*frecv)(struct soap*, char*, size_t); // to save and use recv callback
};
int plugin(struct soap *soap, struct soap_plugin *plugin, void *arg);
```

Then, we write the plugin registry function and the callbacks:

```
#include "plugin.h"
static const char plugin_id[] = PLUGIN_ID; // the plugin id
static int plugin_init(struct soap *soap, struct plugin_data *data);
static int plugin_copy(struct soap *soap, struct soap_plugin *dst, struct soap_plugin *src);
static void plugin_delete(struct soap *soap, struct soap_plugin *p);
static int plugin_send(struct soap *soap, const char *buf, size_t len);
static size_t plugin_recv(struct soap *soap, char *buf, size_t len);
// the registry function:
int plugin(struct soap *soap, struct soap_plugin *p, void *arg)
{
  p-¿id = plugin_id;
  p-¿data = (void*)malloc(sizeof(struct plugin_data));
  p-¿fcopy = plugin_copy;
  p-¿fdelete = plugin_delete;
  if (p-¿data)
    if (plugin_init(soap, (struct plugin_data*)p-¿data))
    {
      free(p-¿data); // error: could not init
      return SOAP_EOM; // return error
    }
  return SOAP_OK;
}
static int plugin_init(struct soap *soap, struct plugin_data *data)
{
  data-¿fsend = soap-¿fsend; // save old recv callback
  data-¿frecv = soap-¿frecv; // save old send callback
  soap-¿fsend = plugin_send; // replace send callback with new
  soap-¿frecv = plugin_recv; // replace recv callback with new
  return SOAP_OK;
}
// copy plugin data, called by soap_copy() // This is important: we need a deep copy to avoid data
sharing by two run-time environments
static int plugin_copy(struct soap *soap, struct soap_plugin *dst, struct soap_plugin *src)
{ *dst = *src;
}
// plugin deletion, called by soap_done()
static void plugin_delete(struct soap *soap, struct soap_plugin *p)
{ free(p-¿data); // free allocated plugin data
}
// the new send callback
static int plugin_send(struct soap *soap, const char *buf, size_t len)
{
  struct plugin_data *data = (struct plugin_data*)soap_lookup_plugin(soap, plugin_id); // fetch
plugin's local data
  fwrite(buf, len, 1, stderr); // write message to stderr
  return data-¿fsend(soap, buf, len); // pass data on to old send callback
}
// the new receive callback
static size_t plugin_recv(struct soap *soap, char *buf, size_t len)
{
  struct plugin_data *data = (struct plugin_data*)soap_lookup_plugin(soap, plugin_id); // fetch
plugin's local data
  size_t res = data-¿frecv(soap, buf, len); // get data from old recv callback
```

```
  fwrite(buf, res, 1, stderr);
  return res;
}
```

The fcopy and fdelete callbacks of **struct** soap_plugin MUST be set to register the plugin. It is the responsibility of the plug-in to handle registry (init), copy, and deletion of the plug-in data and callbacks.

The example plug-in should be used as follows:

```
struct soap soap;
soap_init(&soap);
soap_register_plugin(&soap, plugin);
...
soap_done(&soap);
```

Note: soap_register_plugin(...) is an alias for soap_register_plugin_arg(..., NULL). That is, it passes NULL as an argument to plug-in's registry callback.