DPWS Toolkit Version 2

User Guide

V 2.9 - October 25, 2005

Document history

Version	Date	Editor	Comments
0.9	24/09/2004	Antoine Mensch	Initial version covering WS-Addressing extensions added to the gSOAP toolkit
0.96	01/10/2004	Antoine Mensch Stéphane Rouges	§ MULTI note added. § E, E.1.2.2 API changed and completed (struct wsa_endpoint_ref, dpws_serve, dpws_init, dpws_server_init…) § F.1 Developing a multi-services application: namespaces are now automatically set.
1.9	20/10/2004	Antoine Mensch	Second version covering WS-Discovery Main changes appear in sections 4, 6 and 7.
1.95	05/11/2004	Antoine Mensch	Completed second version.
1.96	30/11/2004	Stéphane Rouges	API modified to take into account: - Two-level lookup (device + services), - WS-MetadataExchange information retrieval
1.97	09/01/2005	Stéphane Rouges	WS-Eventing support induced serveral API modifications. One of the most important is that the user is not in charge of writing a dispatch function anymore, but he must register the handling functions.
2.0	28/01/2005	Stéphane Rouges	API functions *dpws_stop_server* and *dpws_check_device_type* and several utilities were added.
2.1	18/04/2005	Stéphane Rouges	Features added for DPWS toolkit version 1.1
2.9	25/10/2005	Stéphane Rouges	New toolkit major version: 1. Multi-device, 2. New API, 3. API Reference extracted from this document and is now generated from source code.

Table of contents

A. Introduction

This document is a short user guide for the DPWS toolkit, which implements the DPWS (Devices Profile for Web Services) specification [DPWS] on top of the gSOAP open-source SOAP toolkit [gSOAP]. It covers the general principles of programming with gSOAP, as well as the specific extensions to gSOAP introduced to support DPWS. For details on gSOAP programming, the reader is referred to the gSOAP user guide [gSOAP Guide].

This document is work in progress. Any feedback on unclear or missing information is welcome.

A.1 Scope

The DPWS toolkit provides extensive support of the complete DPWS specification [DPWS]. Only some optional features are not supported in this version. The available features, covered in this guide, include:

- SOAP 1.2 support: this is a feature of the gSOAP toolkit.
- WS-Addressing: the DPWS toolkit provides a complete support for the WS-Addressing [WS-Addressing] specification, which is required by the DPWS specification.
- WS-Discovery: the DPWS toolkit provides support for the WS-Discovery [WS-Discovery] specification. Only the optional discovery proxies are not covered by the implementation.
- WS-MetaDataExchange: A partial support of this specification [WS-MetaDataExchange] is provided. For instance, metadata information for a device or service is not extensible and WSDL or schema information are only returned by reference.
- WS-Eventing: the DPWS toolkit provides a complete support for the WS-Eventing [WS-Eventing] specification.

The intended audience of this document is:

- software architects,
- software developers.

It is assumed that the reader has a working knowledge of Web Services, and in particular the Web Services Description Langage [WSDL] and the SOAP protocol [SOAP].

A.2 Abbreviations

DPWS: Devices Profile for Web Services

HTTP: HyperText Transfer Protocol

WSDL: Web Services Description Language

XML: eXtensible Markup Language

A.3 Terms and Definitions

Proxy (or stub) : a software module (function or class) that provides transparent access to a remote function in a client program, by transforming a proxy function call into a network message.

Skeleton : a software module used in a server program to invoke a remote function upon reception of a network message.

Marshalling/demarshalling (or unmarshalling): the process of transforming an in-memory software structure into a network-friendly message representation, and the reverse process.

SOAP : a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment.

Web Service: a service that is accessible from remote clients using Web-based technologies such as HTTP and SOAP.

A.4 Document Guide

This document is organised as follows:

- This section and the following one provide context information and references used in the document.
- The third section is a high-level description of the target DPWS architecture. It is useful to gain an understanding of the organisation of predefined and user-defined services in a DPWS-enabled device.
- The following section presents the main principles of Web Services development using the DPWS toolkit. It describes the various artefacts that are involved in the creation of a service.
- The following section is a quick tutorial on how to get started with the DPWS toolkit.
- The last section is intended for advanced topics. In this second version, it covers only the steps that are required to build a multi-services application.

As a convention, the following namespace prefixes will be used when refering to XML content:

- SOAP-ENV: *http://www.w3.org/2003/05/soap-envelope* , the namespace for SOAP 1.2 envelopes.
- SOAP-ENC: *http://www.w3.org/2003/05/soap-encoding* , the namespace for SOAP 1.2 encoding.
- xsi: *http://www.w3.org/2001/XMLSchema-instance* , the namespace for dynamic XML schema informations.
- xs: *http://www.w3.org/2001/XMLSchema* , the namespace for XML schema declarations.
- wsa: *http://schemas.xmlsoap.org/ws/2004/08/addressing* , the namespace for WS-Addressing headers.
- wdp: *http://schemas.xmlsoap.org/ws/2005/05/devprof* , the namespace for DPWS elements.
- wsd: *http://schemas.xmlsoap.org/ws/2005/04/discovery*, the namespace for WS-Discovery headers and elements.
- wsp: *http://schemas.xmlsoap.org/ws/2002/12/policy*, the namespace for WS-Policy elements.
- wsm: *http://schemas.xmlsoap.org/ws/2004/09/mex*, the namespace for WS-MetadataExchange elements.
- wse: *http://schemas.xmlsoap.org/ws/2004/08/eventing*, the namespace for WS-Eventing elements.

Note that the above prefixes are only defined here for convenience and are not significant without a proper declaration in a XML document.

B. References

[SOAP] : *SOAP Version 1.2 Part 1: Messaging Framework* (http://www.w3.org/TR/soap12-part1/), M. Gudgin, M. Hadley, N. Mendelsohn, J-J. Moreau, H. Frystyk Nielsen, Editors. World Wide Web Consortium, June 24, 2003.

[WSDL]: *Web Services Description Language (WSDL) 1.1* (http://www.w3.org/TR/2001/NOTE-wsdl-20010315), E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, Authors. World Wide Web Consortium, March 15, 2002.

[WS-I]: *Basic Profile Version 1.0* (http://www.ws-i.org/Profiles/BasicProfile-1.0.html). The Web Services-Interoperability Organization, April 16, 2004.

[WS-Addressing]: *Web Services Addressing (WS-Addressing), W3C Member Submission.* (http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/).

[WS-Discovery]: *Web Services Dynamic Discovery (WS-Discovery)*, Web Services Specification (http://schemas.xmlsoap.org/ws/2005/04/discovery/), Microsoft, BEA and Canon, April 2005.

[WS-Policy] : *Web Services Policy Framework (WS-Policy)*, Web Services Specification, (http://msdn.microsoft.com/ws/2002/12/Policy/), BEA, IBM, Microsoft and SAP, May 28, 2003.

[WS-Metadata] : *Web Services Metadata Exchange (WSMetadataExchange)*, Web Services Specification, (http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-metadataexchange.pdf), /), BEA, IBM, Microsoft, Sun, CA and SAP, September 2004.

[WS-Eventing]: *Web Services Eventing (WS-Eventing)*, Web Services Specification, (http://msdn.microsoft.com/ws/2004/08/ws-eventing/), BEA, IBM, Microsoft, Sun, CA and TIBCO, August 2004.

[DPWS]: *Devices Profile for Web Services* (http://schemas.xmlsoap.org/ws/2005/05/devprof/). Microsoft, May 2005.

[gSOAP]: *SOAP C/C++ Web Services* (http://www.cs.fsu.edu/~engelen/soap.html). Note that this site contains the latest version of the gSOAP documentation, which is not the one the DPWS toolkit is built upon.

[gSOAP Guide] : *gSOAP 2.7.6 User Guide* , R. van Engelen. Genivia Inc., September 2, 2005. Available in the DPWS distribution package.

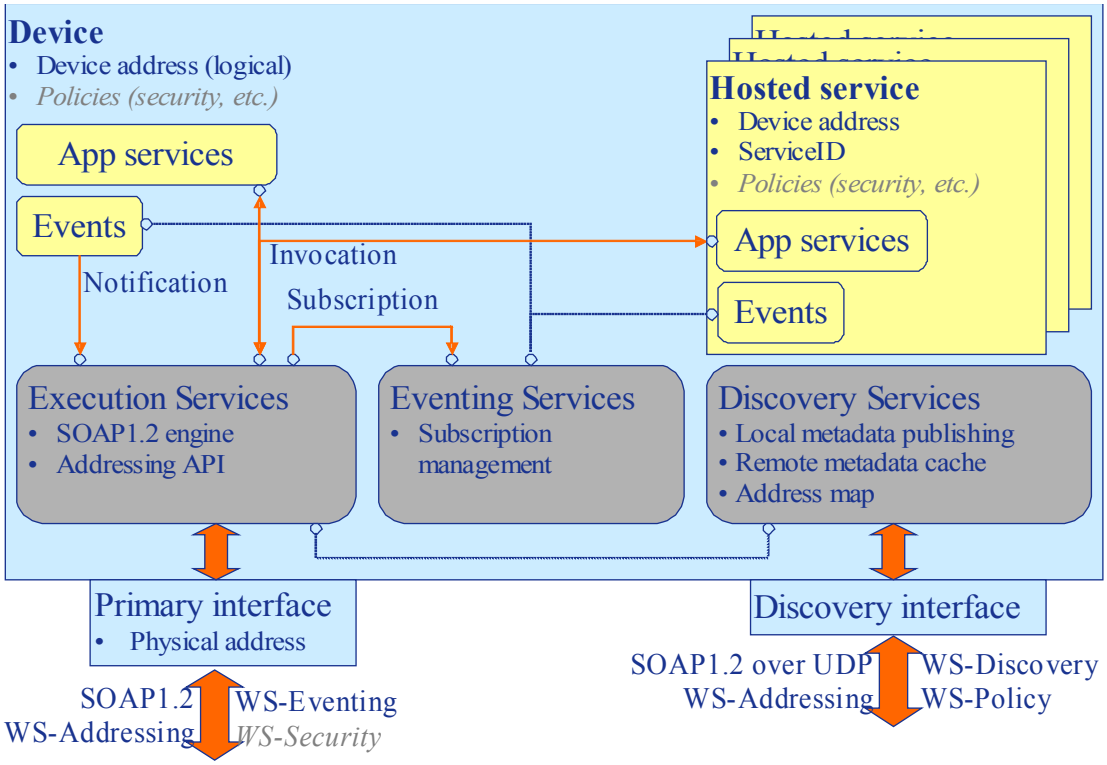
C. Architecture principles

The DPWS specification defines an architecture in which devices run two types of services: hosting services and hosted services. Hosting services are directly associated to a device, and play an important part in the device discovery protocol. Hosted services are mostly functional and depend on their hosting device for discovery. In addition to those user-defined services, DPWS specifies a set of built-in services:

- Discovery services (WS-Discovery): those services are used by a device connected to a network to advertise itself and to discover other devices.
- Metadata exchange services (WS-MetadataExchange): those services provide dynamic access to a device's hosted services and to their metadata, such as WSDL or XML Schema definitions
- Event publish/subscribe services (WS-Eventing): those services are extensions of user-defined services, and allow other devices to subscribe to asynchronous messages (events) produced by a given user-defined service.

DPWS is built on top of the SOAP 1.2 standard, and relies on additional Web Services specifications, such as WS-Addressing and WS-Policy, to further constrain the SOAP messaging model.

The following figure shows the general architecture of a device compliant with DPWS:



DPWS-compliant device architecture

In the above figure:

- User-defined services and events are shown in yellow boxes. They are provided as user-written code and generated code in the DPWS toolkit.
- Predefined services, including the embedded SOAP 1.2 engine, are shown in dark-grey boxes. They are provided as run-time libraries in the DPWS toolkit.

- The two network interfaces are shown: the primary interface uses the standard SOAP 1.2 over HTTP binding to exchange regular SOAP messages, while the discovery interface uses UDP and a multicast address to broadcast and listen to the predefined discovery messages. Both interfaces rely on a standard IP stack

Note: the current version of the DPWS toolkit provides the following functionality:

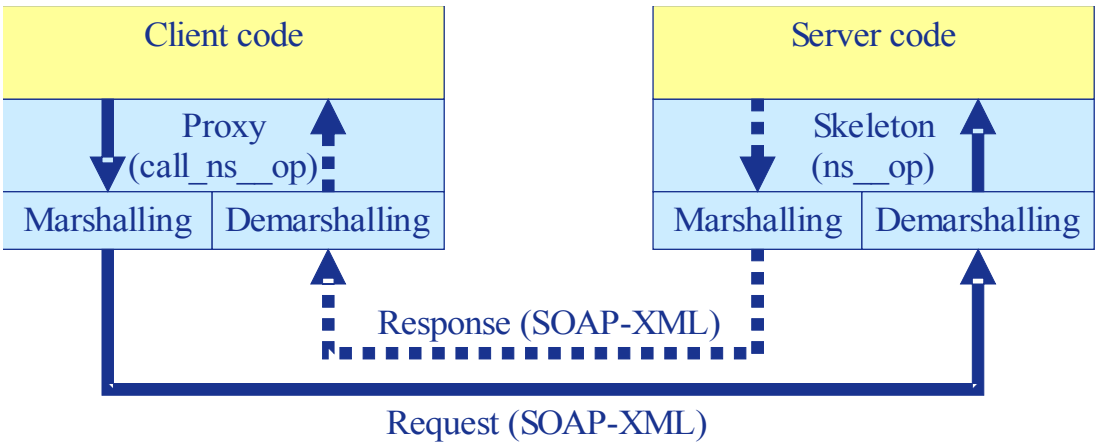
- service invocation using the SOAP 1.2 engine and the WS-Addressing specification,
- automated discovery using WS-Discovery mechanisms,
- partial support for WS-MetaDataExchange,
- event management through the support of WS-Eventing.

D. gSOAP and DPWS development principles

The DPWS toolkit uses the same approach as the gSOAP toolkit, namely it relies heavily on code generation to automatically map back and forth SOAP-XML messages and C/C++ structures. Starting from a service definition, which involves operations and data types used as parameters and return values to those operations, the toolkit generates the required code to provide transparent access to the remote operations from a client. The only pieces left to implement by the developer are:

- The implementation of the operations in the server.
- The implementation of the client that invokes the remote operations.

Those principles are highlighted in the following figure:



Remote operation invocation in gSOAP

Proxy, skeleton, marshalling and demarshalling (i.e. the transformation between C/C++ structures and SOAP-XML messages) code is completely generated by the toolkit. The marshalling/demarshalling code is the same in the client and the server.

SOAP responses are returned only when using the request/reply message exchange pattern. In case of one-way messages, only the request is transmitted.

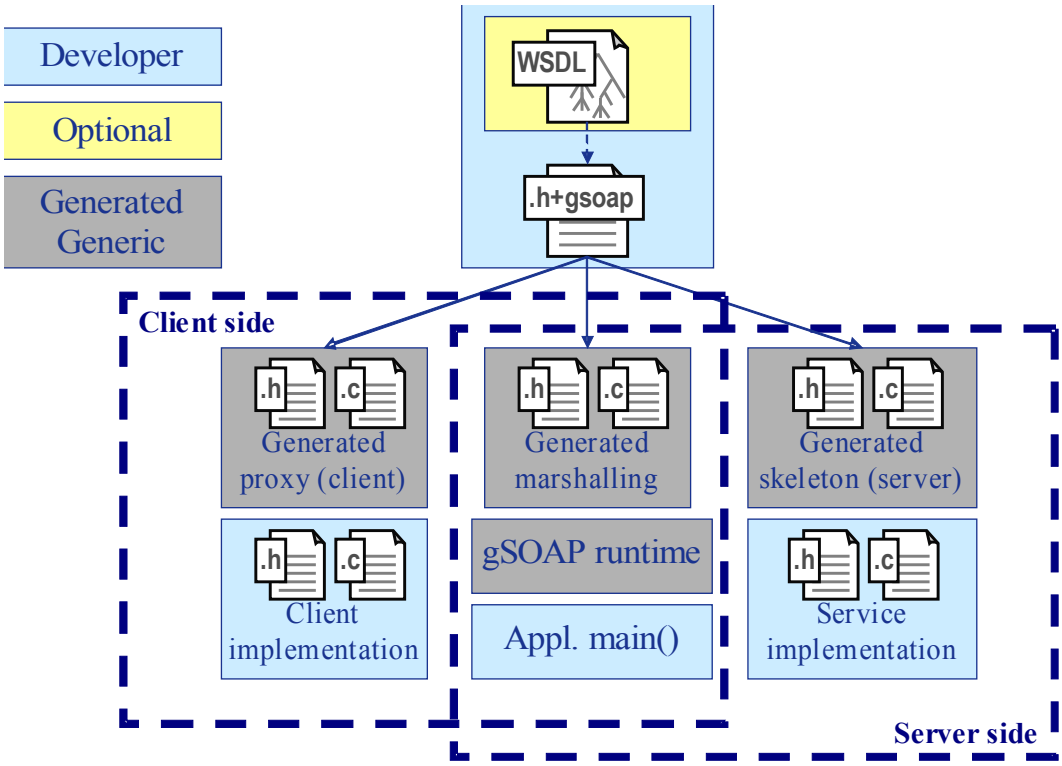
The gSOAP toolkit uses an annotated C/C++ header file as its main service definition. This file contains:

- C structs and C++ classes declarations and typedefs,
- Prototypes of the remote operations,
- Annotations controlling the way structures and operations are mapped into SOAP-XML messages.

This annotated header file is not a standard C/C++ header file, and must be processed by the gSOAP compiler to produce valid C/C++ files. This compilation process also produces an equivalent WSDL document describing the generated service.

gSOAP also provides a second code generator that transforms a WSDL document into an annotated header file ready for processing by the gSOAP compiler. Although useful in simple cases, the use of WSDL files does not provide the same level of flexibility as the use of the annotated header files for developing modular, multi-services applications. For this type of applications, direct modifications of the annotated header files might be required.

The following figure summarises the various artefacts involved in Web service development, using gSOAP:



gSOAP code generation principles

The DPWS toolkit extends the above code generation principles to take into account new requirements induced by the implementation of the WS-Addressing specification. This specification states that a SOAP response and fault message may be redirected at user choice to an endpoint which is not necessarily the origin of the request. This means that the standard, synchronous approach, which uses the HTTP response to carry the response or fault message associated to a SOAP request sent through an HTTP request, is not always applicable. The DPWS toolkit thus supports asynchronous transfer of response or fault messages to a specified endpoint. Such an endpoint should be set up as a message server, and should invoke a user-defined handler upon asynchronous reception of a SOAP response or fault message. In order to ease the development of those handlers, the DPWS toolkit generates header and C files implementing the skeletons for the handlers, following the pattern used by the gSOAP service skeleton generation. A similar approach is used for handling events: endpoints that subscribe to events should be set up as message servers, and use the event handling skeletons generated by the DPWS toolkit to process received events.

The initialisation of a Web Services server (i.e. a device) is also impacted by the DPWS specification: this specification defines discovery messages used by devices to exchange information about service location. Those messages are automatically generated by a device, based on start-up configuration information, including:

- Device identifier which remains stable across reboots.
- HTTP address for the server.
- Target namespaces and port types of supported services.
- Metadata version number.

The initialisation code of a device must therefore fill up the appropriate structures to configure the discovery mechanism.

E. Getting started

Instructions given in this chapter are guide-lines for building a web service implementation and invocation. For concrete use cases, please refer to the samples in the delivery package.

E.1 Generating proxies and skeletons

E.1.1 From a WSDL file

This approach is well-suited for simple web service invocation and implementation, especially when one does not want to learn the gSOAP annotation language. This is an optional step, since proxies and skeletons will anyhow have to be generated from an annotated header file, that can be generated with the tool described in this paragraph or written directly by hand.

The approach consists in generating an annotated header file from a standard WSDL file, as described in the gSOAP documentation [gSOAP Guide], § 8.2.10. For example, the following command would produce an annotated *myService.h* header file :

```
wsdl2h –c myService.wsdl
```

The '-c' option is used to generate pure C header file. The generated file will then be processed by the gSOAP stub & skeleton code generator, as described in the next paragraph.

E.1.2 From an annotated header file

The annotated header file is the 'native' source file for the gSOAP code generator (also called the "stub & skeleton compiler" in gSOAP). This is the preferred way to specify a web service in advanced use cases since it covers the whole set of gSOAP functionalities. This approach should be used for instance when one needs to implement multi-services servers, since the gSOAP instructions enables multiple services merging by special directives. To learn about the gSOAP stub & skeleton compiler, please refer to [gSOAP Guide] §9.

gSOAP annotated header files are made up of :

- C function prototypes. They must be specified following the rules described in [gSOAP Guide] §10.
- C/XML Schema type mapping specifications. A standard mapping is described in [gSOAP Guide] §11, but the user can also define a custom one.
- gSOAP directives that are mainly used for WSDL generation. Indeed, when the previous step (generation from a WSDL file) has not been performed, the gSOAP compiler also generates a WSDL file based on the provided header file. Some directives should be included even in the simplest header file, like namespace declarations for instance, to avoid specifying them by hand later on. For a complete description of the gSOAP directives, please refer to the gSOAP documentation ([gSOAP Guide] §19.2).

E.1.2.1 DPWS Custom directives

The gSOAP annotation language has been extended with two new directives to support WS-Addressing requirements, namely the inclusion of a wsa:Action header associated to each input or output message involved in a given WSDL operation:

```
//gsoap namespace-prefix service method-request-action:   wsdlOperation wsaAction
```

```
//gsoap namespace-prefix service method-response-action:  wsdlOperation wsaAction
```

Two new directives have also been added to support WS-Eventing and allow the user to specify event sources:

- The first one is used to specify that the service (or, more formally, the port type implemented by the service) must be considered as an event source, in the sense of WS-Eventing:

```
//gsoap namespace-prefix service event-source: true
```

- The second one identifies specific operations as output (when no return value is specified) or output/input messages.

```
//gsoap namespace-prefix service method-message-exchange-pattern: wsdlOperation output
```

Note:

When generating the annotated header file from a WSDL file, those DPWS custom directives are automatically inserted in the generated file.

E.1.2.2 Remote and generated functions prototypes

For a given operation, function prototypes are defined as

ns__function(param1, param2, …, paramN, return)

in the gSOAP annotated header file. The final return parameter specifies the type of the value returned by the operation. This should normally be of a pointer type, so that the parameter can be used as output.

Note that a one-way operation can be specified using *void* as a return type ([gSOAP Guide] §8.3). An operation with no input parameters should either use an empty structure as only parameter, or a void* type, when the C compiler does not support empty structures ([gSOAP Guide] §8.1.14). An operation with no output parameter should do the same for the return parameter ([gSOAP Guide] §8.1.15).

E.1.2.3 Generation

To generate stubs & skeletons from an annotated header file use the following command line:

```
soapcpp2 -2uc myService.h
```

- The '-2' must be specified to specify that SOAP 1.2 envelopes should be used (insead of SOAP 1.1 envelopes),
- **The '-u' must be specified to generate the DPWS extensions to gSOAP,**
- The '-c' option is used to generate pure C stubs & skeletons.

Multiple files are generated. Among those:

- *soapC.c, soapH.h* contains marshalling/demarshalling code,
- *soapStub.h* contains stub & skeleton declarations (derived from the information of the annotated header file),
- *soapServer.c* contains the skeleton code that handles SOAP and DPWS processing and calls the user service implementation.
- *soapClient.c* contains the stub code.

- *soapHandler.c* contains the code for handling asynchronous responses that may have been initiated by a distinct endpoint and for events. This file is a DPWS (in fact WS-addressing) specific add-on.
- *[serviceName].nsmap* contains prefix definitions for the web service.
- *soapServerLib.c, soapClientLib.c* and *soapHandlerLib.c* are versions of stubs & skeletons that include a 'static' version of the marshalling/demarshalling code that contribute to enable multiple web service use in a single program.
- Various XML files providing SOAP messages examples (that do not include WS-addressing headers).
- Other header files are used for C++ implementation/invocation of web services.

E.2 Developing a DPWS client

E.2.1 Programming

Writing a client program using the stub generated at the preceding step is quite simple. To show this, let's walk through the necessary instructions of a C client program:

```
#include "gen/soapStub.h"
#include "gen/calc.nsmap"
```

These include directives declare the available stub functions and types, as well as the namespace prefix definitions. These two include files are generated by the gSOAP compiler.

```
struct dpws dpws;
struct wsa_endpoint_ref * devices[3],* endpoints[3];
int nEndpoints = 0;
struct model_info * model;
```

The DPWS API relies on a single data structure, *dpws*. It will be supplied to every API function and encapsulates the *soap* structure used by the gSOAP runtime. The *dpws* structure is defined in the *dpws.h* header file. Direct access to the structure is useful to read or write WS-Addressing header information. The array of pointers to endpoint reference (*wsa_endpoint_ref* structure, defined in the *dpws.h* header file) is used to retrieve the endpoints implementing a given service.

```
dpws_init();
```

This function is to be called before any other call to the DPWS API for initialization purposes.

```
dpws_client_init(&dpws, NULL );
```

This function is to be called once to initialize the *dpws* data structure. The second parameter can be null and may contain an endpoint reference structure that will be passed as the WS-Addressing source endpoint (wsa:From header) in every SOAP message.

```
nEndpoints = dpws_lookup(&dpws, "http://myDevice", "myDeviceType", NULL, devices, 3);
```

This function is used to retrieve one or several device endpoint references supporting the specified type in the specified WSDL namespace (http://myDevice) and in the specified scope (in this case, the NULL value indicates that the scope is not constrained). To achieve this the function relies on [WS-Discovery] messages or on an internal cache if the information was already retrieved. The retrieved endpoints are stored in the *devices* array. The last parameters represents the maximum number of endpoints to be returned. The function returns the actual number of retrieved endpoints, or –1 if an error occurred. Note that performing the endpoint lookup before each call means that the client will be reactive to any change of configuration in the network, at the expense of slightly reduced performances (available endpoints are cached and updated automatically using the WS-Discovery protocol). On the other hand, using dynamic lookup is not a

guarantee that a returned endpoint is running, as cache update mechanisms are not instantaneous. Thus, the client code should be ready to repeat the lookup and call operations if a returned endpoint is not able to handle the call.

Once device are retrieved, the user can do two things. Either get device metadata or retrieve the device hosted services endpoints. To get metadata, a set of functions are available (see API reference). For instance, to get information on the device model:

```
model = dpws_get_model_info(&dpws, device[0]);
```

The second parameter is any valid device endpoint, preferably retrieved by a preceding *dpws_lookup* call. Note that the structure returned is allocated dynamically on the gSOAP heap and will remain available until *dpws_end* is called on the *dpws* runtime structure.

To get service endpoints hosted by the first device retrieved before:

```
nEndpoints = dpws_get_service(&dpws, device[0], "http://myService", "myServiceType",
                          endpoints, 3);
```

This call is quite similar to *dpws_lookup*, except it applies on a specified device and thus does not need scope. Indeed, it corresponds (when information is not in the cache) to a [WS-Metadata] *getMetadata* message performed over a connected (TCP) communication.

```
if (nEndpoints > 0)
    dpws_call_ ns __ operation (&dpws, endpoints[0] ,NULL,p1,p2,…,&result);
```

This line performs the remote call. Note that:

- The second parameter is a pointer to the *wsa_endpoint_ref* structure representing the endpoint that hosts the web service.
- The third parameter is an optional pointer to a *wsa_endpoint_ref* structure representing the endpoint to which the reply should be sent. If NULL, the call will be performed synchronously and the reply will be sent to the current program. If a structure is supplied, then the remote call will return immediately, without waiting for the response. **A handler is then supposed to be implemented on the "reply to" endpoint to receive and process the response**.
- The following parameters are the input parameters of the remote call,
- The last parameter will contain the result of the call. It may be missing if the web service has no return value.

See the *soapStub.h* file for a complete description of the remote call.

In case of failure (that can be also detected by testing the return value of the remote call), you may want to print an error message :

```
if (dpws_fault(&dpws) )
    dpws_print_fault(&dpws, stderr);
```

```
dpws_end(&dpws);
```

This last command will release memory allocated dynamically for the SOAP messages processing. This should be called after every invocation but remember that is may also release the endpoints retrieved through *dpws_lookup* .

For more details on how to build a client program using gSOAP, please refer to [gSOAP Guide] §8.1.

E.2.2 Client-side functions prototypes

The stub function used on the client side to invoke the remote function has the following prototype:

dpws_call_ns__function(struct dpws*, struct wsa_endpoint_ref* to, struct wsa_endpoint_ref* reply_to, param1, param2, …, paramN, return)

The stub function expects a mandatory *to* endpoint reference parameter that represents the destination of the call.

An optional *reply_to* endpoint reference parameter can be given to specify that the call should return immediately without filling the output parameter, and that the response should be sent asynchronously to the specified endpoint. Note that such an endpoint should have set up the appropriate handler dispatch function to properly process the returned value. When this parameter is NULL, the call will wait for the response to be returned in the output parameter.

The additional parameters correspond to the remote function parameters. The corresponding parameters will be absent if an empty structure type has been specified as input or output parameter. Notice that the user has to allocate and initialize these parameters (for non basic C types). If allocation is performed using gSOAP *soap_malloc* primitive, data will be automatically freed during the call to *dpws_end*. The *soap_defaut_[type]* serialization methods can be used to initialize complex structures (see [gSOAP Guide] for more information).

When using a one-way operation, the proxy function used on the client side to invoke the remote function has the following prototype:

dpws_send_ns__function(struct dpws*, struct wsa_endpoint_ref* to, param1, param2, …, paramN)

The stub function expects a mandatory *to* endpoint reference parameter that represents the destination of the call.

The additional parameters correspond to the remote function parameters. The corresponding parameters will be absent if an empty structure type has been specified as input parameter. Note that the socket used to send the message is not closed in one-way interaction. This can be automatically performed by calling the *dpws_end* function at the end of the invocation code.

E.2.3 Building

To generate a client executable, you must compile and link your program with the following files:

- *soapClient.c* (generated by the DPWS/gSOAP compiler), the stub,
- *soapC.c* (generated by the DPWS/gSOAP compiler), the marshalling/demarshalling code,
- the DPWS runtime, which includes the gSOAP runtime (note: the gSOAP runtime *stdsoap2.c* must be on the include path):
  - cache.c,
  - config.c,
  - discovery.c,
  - dpws.c,
  - event.c,
  - mdm.c,
  - metadata.c,
  - my_stdsoap2.c,

– subscmanager.c,

– tools.c.

E.3 Developing a DPWS server

A DPWS server code is made of three main pieces of code:

1. The server configuration,
2. The server loop,
3. The service operations.

E.3.1 Server configuration objects

Server configuration uses an object model where objects are referenced by handles. The configuration objects for a server are of the following types:

- A special unique object (a singleton) for ***toolkit configuration (1)*** . Note that a special handle ref is reserved for this one (`TOOLKIT_HANDLE`),
- ***Device classes (2)*** are device types that will share properties like characteristics (e.g. model information) or *service classes*,
- ***Service classes (3)*** are service types that will share properties like port types,
- ***Device instances (4)*** are operational DPWS device entities which belong to a *device class*,
- ***Service instances (5)*** are operational DPWS hosted services entities which belong to a *service class.*

The user can create these objects with the APIs described in this chapter and set objects attributes (i.e. Properties) when applicable on the object type.

E.3.2 Configuration objects attributes

The following table describes the server object configuration attributes, especially on which object type it is applicable and if it is a multivalued attribute.

Parameter name	*C Type*	*Default*	*Applic. on*	*Multi*	*Description*
`DPWS_STR_MAC_ADDR`	char*	Retrieved	1	No	6 bytes given as hexadecimal strings separated by '-' or ':'. Can be retrieved from the network interface on platforms that support it.
`DPWS_STR_IP_ADDR`	char*	Retrieved	1	No	IP address in the xxx.xxx.xxx.xxx notation. Can be retrieved from the network interface on platforms that support it.
`DPWS_INT_BOOT_SEQ`	int	-	1	No	The boot sequence must be incremented each time the device connects to the network, and should probably use some non-volatile memory mechanism to work properly.
`DPWS_STR_HTTP_ADDR`	char*	Calculated	1	No	URL of on which the toolkit is listening.
`DPWS_INT_HTTP_PORT`	int	80	1	No	Port on which to bind the listening TCP socket
`DPWS_INT_HTTP_BACKLOG`	int	10	1	No	Listening TCP socket backlog
`DPWS_PTR_CALLBACK_HELLO`	void(*)(struct wsa_endpoint_ref *)	NULL	1	No	User callback called when a *Hello* message is received. The parameter passed to is the *uuid* of the device (which can be local).
`DPWS_PTR_CALLBACK_BYE`	void (*)(struct wsa_endpoint_ref *)	NULL	1	No	User callback called when a *Bye* message is received. The parameter passed to is the *uuid* of the device (which can be local).
`DPWS_INT_DEFAULT_DEVICE`	short	-1	1	No	Device used during message routing when the address of the message is the toolkit HTTP address (no endpoint address).
`DPWS_STR_PREFERRED_LANG`	char *	NULL	1	No	Used to select a value when multiple language are available through the XML "lang" attribute (in WS-MetadataExchange and WS-Eventing)
`DPWS_STR_MAX_SUBSC_DURATION`	char *	"P1D"	1-5	No	The maximum duration that the device grants to

					suscribers. See [WS-Eventing].
DPWS_BOOL_MANAGEMENT_SERVICE	int	0 (FALSE)	1-2	No	Enables the Metadata Manager built-in web service.
DPWS_PTR_TYPE	struct qname *	NULL	2-3	yes	A WSDL port type attached to the device or service class. As no prefix is specified, one will be generated for outgoing messages.
DPWS_PTR_PREFIXED_TYPE	struct prefixed_qname *	NULL	2-3	yes	A WSDL port type (with a prefix for outgoing messages) attached to the device or service class.
DPWS_PTR_WSDL	struct wsdl_info *	NULL	2-3	yes	A WSDL file information attached to the device or service class.
DPWS_PTR_HANDLING_FUNCTION	int (*)(struct dpws*)	NULL	2-3	Yes	A message dispatch function generated by the gSOAP/DPWS compiler that will route the message to the matching processing function.
DPWS_PTR_FAULT_FUNCTION	int (*)(struct dpws*)	NULL	2-3	No	A user function that is called by the DPWS server loop when asynchronous call led to a fault. The user can then check the WSA header contents and call the *dpws_get_fault()* and *dpws_print_error_msg()* API calls.
DPWS_PTR_CALLBACK_EVENT_END	void (*) (struct event_end_info *)	NULL	2-3	No	A user function that is called by the DPWS server loop when an abnormal WS-Eventing event end notification is received by an endpoint of the corresponding class.
DPWS_STR_ID	char*	generated	2-3	No	An alphanumeric ID that may be used to identify the object instead of the numerical handle reference. The default value is the string form of the object handle reference
DPWS_STR_MODEL_NAME	char*	-	2	No	Device characteristic metadata [DPWS]: *user-friendly name for this model of device chosen by the manufacturer*.
DPWS_STR_MODEL_NUMBER	char*	NULL	2	No	Device characteristic metadata [DPWS]: *model number for this model of device*.
DPWS_STR_MODEL_URL	char*	NULL	2	No	Device characteristic metadata [DPWS]: *URL to a Web site for this model of device.*
DPWS_STR_PRESENTATION_URL	char*	NULL	2	No	Device characteristic metadata [DPWS]: *URL to an HTML page for this device.*
DPWS_STR_MANUFACTURER	char*	-	2	No	Device characteristic metadata [DPWS]: *the name of the manufacturer of the device.*
DPWS_STR_MANUFACTURER_URL	char*	NULL	2	No	Device characteristic metadata [DPWS]: *URL to a Web site for the manufacturer of the device.*
DPWS_INT_METADATA_VERSION	int	-/ generated	2-4	No	The metadata version must be incremented each time the DPWS metadata changes, as it is used to control cache obsolescence in clients of the device. This attribute is mandatory for device class because its management is delegated to the API user who is best aware of persistent changes in the device. For the deviceS instance, metadata version may be managed automatically by the toolkit or set by the API user.
DPWS_STR_ADDRESS	char*	generated	5	No	A relative or absolute URL that will constitute the physical address of the hosted service endpoint. If relative (e.g. "MyServices/123457"), it will be appended to the toolkit HTTP address.
DPWS_STR_SERVICE_ID	char *	service class id	5	No	An alphanumeric ID used to identify the service and is returned in device metadata.
DPWS_STR_DEVICE_ID	char*	generated	4	No	Stable device ID. Can be generated as a UUID from the MAC address and listening port if not specified. If specified, must be of the form "uuid:cf72487e-3594-11da-a988-000f1f26b50d"
DPWS_PTR_SCOPE	char*	Cf. Spec.	4	Yes	A scope URI [DPWS].
DPWS_STR_FRIENDLY_NAME	char*	-	4	No	Device characteristic metadata [DPWS]: *user-friendly name for this device*.
DPWS_STR_FIRMWARE_VERSION	char*	NULL	4	No	Device characteristic metadata [DPWS]: *firmware version for this device.*
DPWS_STR_SERIAL_NUMBER	char*	NULL	4	No	Device characteristic metadata [DPWS]: *manufacturer-assigned serial number for this device*.
DPWS_PTR_USER_DATA	void *	NULL	4	No	User data, specific to the device instance that can be used to diversify

NOTE :

Attributes that do not have default values (-) are **mandatory**.

E.3.3 Programming

Here is the structure of a simple DPWS server main function:

```
#include "gen/soapStub.h"
#include "gen/calc.nsmap"
```

As for the client side, these include directives declare the available skeleton functions and types, as well as the namespace prefix definitions. These two files are generated by the gSOAP compiler.

```
struct dpws    dpws, dpws_tasks[MAX_THREADS];
short href_dev_class, href_serv_class, href_dev, href_serv;
char* scopes[] = { "urn:scope1", "urn:scope2" };
struct qname deviceType = {"http://myDevice", "deviceType"};
struct qname portTypes[] =     {{"http://myService1", "portType1"},
                               {"http://myService1", "portType2"}};
struct wsdl_info wsdl = {"http://myService1",
                    "http://www.schneider-electric.com/myService1.wsdl"};
```

The first *dpws* structure will be used for listening on the socket and could be enough for a mono-thread application. The array of *dpws* structures is intended for a multi-thread processing of requests. The *scopes* array of strings is a constant used to initialise the device at startup (see below). *DeviceType structure, portTypes1* and *portTypes2* arrays of qualified qname structures are constants used to initialize device hosted services at startup (see below).

```
dpws_init();
```

This function is to be called before any other call to the DPWS API for initialization purposes.

In order to support the WS-Discovery and WS-MetadataExchange specifications, the server must be initialised with a set of parameters. Some of those parameters are optional (e.g. MAC address, IP address, scopes or listening port number), as they can be retrieved automatically or have meaningful default values, at least on some platforms, while others must be provided.

```
DPWS_SET_INT_ATT (TOOLKIT_HANDLE, DPWS_INT_HTTP_PORT, 8888);
DPWS_SET_INT_ATT(TOOLKIT_HANDLE, DPWS_INT_BOOT_SEQ, IMPLEMENTATION_DEPENDANT_VALUE);
```

Among the two parameters shown above, the first has a default value but is usually specified:

- the http port used for the server web service invocation. The default value is the standard HTTP port. Specify it if for instance the port is already busy with an HTTP server.

The other must be absolutely specified when initializing a full DPWS server:

- The boot sequence must be incremented each time the device connects to the network, and should probably use some non-volatile memory mechanism to work properly,

The first configuration object that must be created is the device class:

```
href_dev_class = dpws_create_device_class();
```

Note that this API returns a numeric handle reference that will be used to reference the object in future calls. A set of functions must be called to set up device characteristics, in order for the device to advertise itself through the WS-Discovery protocol:

```
DPWS_SET_INT_ATT(href_dev_class, DPWS_INT_METADATA_VERSION, 300);
DPWS_ADD_STR_ATT(href_dev_class, DPWS_STR_MANUFACTURER, "Schneider Electric SA");
DPWS_ADD_STR_ATT(href_dev_class, DPWS_STR_MODEL_NAME, "SmartTrap");
```

```
DPWS_ADD_PTR_ATT(href_dev_class, DPWS_PTR_TYPE, &deviceType);
```

The following properties have been set using the above macros:

- The metadata version that must be changed each time the metadata changes, as it is used to control cache obsolescence in clients of the device. The complete list of parameters is given in the API reference section,
- Several device metadata information as specified in the [DPWS] specification,
- The device type (or a port type if the device endpoint hosts itself a service) which is an XML qualified name (a namespace and local name pair).

Then services classes must be created and configured for each kind of service:

```
extern int soap_serve_request(struct soap *);

href_serv_class = dpws_create_service_class();

DPWS_ADD_PTR_ATT(href_serv_class , DPWS_PTR_TYPE, &portTypes[0] );
DPWS_ADD_PTR_ATT( href_serv_class, DPWS_PTR_TYPE, &portTypes[1]);
DPWS_ADD_PTR_ATT(href_serv_class, DPWS_PTR_WSDL, &wsdl);
DPWS_ADD_PTR_ATT(href_serv_class, DPWS_PTR_HANDLING_FUNCTION, &soap_serve_request );
```

The initialization phase must also set up hosted services metadata, in order for the device to advertise the hoted services it supports through the WS-MetadataExchange protocol. All the service information which is related to a WSDL file (and therefore a generation) will be attached to a configuration object named a service class which is created with the *dpws_create_service_class* function. The main properties to attach to the service class are:

- the hosted service port types corresponding to the ones (or a subset) defined in the WSDL used for the generation and that will be implemented by the service endpoint created from the service class,
- optional namespaces and locations for the WSDL files,
- The server function for the service (here *soap_serve_request*), which is generated by the gSOAP/DPWS compiler. This function dispatches the message to the corresponding processing function. The programmer will find the prototype of this function (which may have another name according to generation options) defined in the *soapStub.h* header file ('Remote methods' section).

The created service class must then be attached to the previously created device class. Note that a service class can be attached to multiple devices classes.

```
dpws_register_service_class(href_dev_class, href_serv_class);
```

Now object classes have been created and associated, operational instances (endpoints) will be created and configured, and first, a device instance with the following calls:

```
href_dev = dpws_create_device(1, href_dev_class);
DPWS_ADD_STR_ATT( href_dev, DPWS_STR_SCOPE, scopes[0]) ;
DPWS_ADD_STR_ATT(href_dev, DPWS_STR_SCOPE, scopes[1]);
DPWS_SET_STR_ATT(href_dev, DPWS_STR_FRIENDLY_NAME, "TRAP-5");
```

The first parameter is a numerical *id* used to diversify the device *uuid* (universal unique identifier) and then should be stable. The second is the device class that will determine the new device instance characteristics and especially the hosted services that will be created according to the service classes that were registered to the device class. Note that another API, named *dpws_create_custom_device* creates a "naked" device instance and to which service instances can be registered manually. Some usual or mandatory device properties are set in the example :

- The scopes to which the device belongs. If omitted the default value is the [WS-Discovery] specification one,
- The mandatory friendly name of the device.

When an attribute must be set on a service instance (i.e. hosted service endpoint), for instance the service id (that in our case was initialized with the string form of the service class handle reference), a handle reference must be first retrieved for the object:

```
href_serv = dpws_get_service_handle_by_class(href_dev, href_serv_class);
DPWS_ADD_STR_ATT(href_serv, DPWS_STR_SERVICE_ID, "urn:myService1Id");
```

Once create the operational object, they must be made "visible" on the network and the DPWS server initialized:

```
dpws_enable_device(href_dev);
dpws_server_init(&dpws, NULL);
```

dpws_server_init is to be called once. It both data structures and binds to the listening socket. The second parameter is optional and may contain an endpoint reference structure that will be passed as the WS-Addressing source endpoint (wsa:From header) in every SOAP message.

```
dpws_accept_thr(&dpws, &dpws_tasks[free_task]);
```

This call process the incoming connections and prepares a dpws runtime structure for a thread that will perform request processing.

In the case of a mono-thread server the following alternate statement can be used:

```
dpws_accept(&dpws);  // connection entering
```

The following lines correspond to the request processing that may be processed in a "forked" thread or in the main thread, in a mono-thread context:

```
// request thread processing
dpws_serve(&dpws_tasks[free_task]);
dpws_end(&dpws_tasks[free_task]);
```

The *dpws_serve* function triggers the DPWS/SOAP message processing that will call back the functions implementing the operations. *dpws_end* cleans dynamically-allocated memory and reset internal status.

For more details on how to build a server program using gSOAP, please refer to [gSOAP Guide] §8.2.

E.3.4 Service operation functions prototypes

The user-defined remote function invoked by the skeleton must have the following prototype:

ns__function(struct dpws*, param1, param2, …, paramN, return)

The corresponding parameters will be absent if an empty structure type has been specified as input or output parameter.

Notice that the user has to allocate and initialize the return value using gSOAP *soap_malloc* primitive and the *soap_defaut_[type]* serialization methods (see [gSOAP Guide] for more information).

E.3.5 Building

To generate a server executable, you must compile and link your program with the following files:

- *soapServer.c* (generated by the DPWS/gSOAP compiler), the skeleton,
- *soapC.c* (generated by the DPWS/gSOAP compiler), the marshalling/demarshalling code,

- the DPWS runtime, which includes the gSOAP runtime (note: the gSOAP runtime *stdsoap2.c* must be on the include path):
  - cache.c,
  - config.c,
  - discovery.c,
  - dpws.c,
  - event.c,
  - mdm.c,
  - metadata.c,
  - my_stdsoap2.c,
  - subscmanager.c,
  - tools.c.

E.4 Developing a DPWS handler

In case of an "asynchronous" invocation, that is to say when the "reply to" endpoint is set in the WS-Addressing headers, a program that potentially has not originated the request may have to process the response. This is performed through DPWS handlers. This feature is DPWS-specific and not present in the original version of gSOAP.

E.4.1 Programming

A DPWS handler must be implemented exactly as a server except that a dispatch method (defined in the 'Handler Skeletons' of the *soapStub.h* header file) must be registered specifically.

When no need for a real device is needed (no discovery publication) a program must nevertheless a dummy empty device (with no type nor hosted service) to register the corresponding function, use the following statements:

```
href_serv_class = dpws_create_service_class();
DPWS_ADD_PTR_ATT(href_serv_class, DPWS_PTR_HANDLING_FUNCTION, &soap_handle_event );
href_dev = dpws_create_device(1, href_dev_class);
dpws_enable_device(href_dev);
```

Note:

A handler can also be declared for fault processing following an asynchronous call to a web service, using the DPWS_PTR_FAULT_FUNCTION attribute.

E.4.2 Handler prototypes

The user-defined response handler invoked by the skeleton must have the following prototype:

ns__function_handler(struct soap*, return)

The return parameter will be absent if an empty structure type has been specified as output parameter.

E.4.3 Building

To generate a handler executable, you must compile and link your program with the following files:

- *soapHandler.c* (generated by the DPWS/gSOAP compiler), the skeleton,
- *soapC.c* (generated by the DPWS/gSOAP compiler), the marshalling/demarshalling code,
- the DPWS runtime, which includes the gSOAP runtime (note: the gSOAP runtime *stdsoap2.c* must be on the include path):
  - cache.c,
  - config.c,
  - discovery.c,
  - dpws.c,
  - event.c,
  - mdm.c,
  - metadata.c,
  - my_stdsoap2.c,
  - subscmanager.c,
  - tools.c.

E.5 Using WS-Eventing features

WS-Eventing allow a device to send notifications to a set of registered endpoints. The DPWS compiler generates specific stub and skeletons when the specific custom gSOAP directives have been used (see §E.1.2.1, p.9). Besides a set of API is available to manage subscriptions.

E.5.1 Programming

A DPWS event handler must be implemented exactly as a handler for asynchronous response and the dispatch method that must be provided is the same (the one defined in the 'Handler Skeletons' of the *soapStub.h* header file).

For instance, to register the corresponding function, use the following statement:

```
DPWS_ADD_PTR_ATT(href_serv_class, DPWS_PTR_HANDLING_FUNCTION, &soap_handle_event );
```

Registering a event handler is not enough to receive events. A subscription must be performed using the following API:

struct wsa_endpoint_ref * notify_to, *subsc_manager;

```
char ** duration;
struct wsa_endpoint_ref * notify_to;

*duration = "PT1H";
notify_to = dpws_get_local_endpoint(href_dev);
subsc_manager = dpws_event_subscribe(&dpws, event_source, notify_to, NULL, NULL,
duration);
```

This line performs the subscription. Note that:

- The second parameter is a pointer to the *wsa_endpoint_ref* structure representing an event source endpoint. This endpoint reference may have been retrieved using the *dpws_lookup* and *dpws_get_service* APIs.
- The third parameter is the endpoint of the service on which the event handler has been registered. Note that in our case this is the device endpoint.
- The fourth parameter is an optional endpoint reference that may receive (is such a feature is required) an event if the event source must cancel subscription before the expected duration is passed. Such an event may be caught on the "endTo" endpoint using the DPWS_PTR_CALLBACK_EVENT_END attribute on the device or service class. Note that the application will have to run a DPWS server loop to receive this event.
- The fifth parameter is a null-terminated URI array that allow to specify which operations of the event source service must be received.
- The last parameter specify the duration of the subscription. Note that a event source may accept only durations (like the 1 hour example) and not absolute dates. The event source may also grant a shorter duration than the one asked (hence the char **).

Other APIs are available to manage the subscription (see API Reference). Each of these will take as input the endpoint reference we named "subsc_manager" in the sample and that contains an unique ID for the subscription.

The events themselves are generated by an event source using the server stubs like described hereafter:

```
source = dpws_get_local_endpoint(href_serv);
dpws_notify_ ns __ operation (&dpws, source , p1, p2,…);
```

This line performs the remote call. Note that:

- The second parameter is a pointer to the *wsa_endpoint_ref* structure representing the service endpoint that is the event source. Indeed multiple services may implement a same port type on a DPWS device.
- The message parameter follow according the definition of the operation.

E.5.2 Event handler prototypes

The user-defined event handler invoked by the skeleton must have the following prototype:

ns__function (struct soap*, param1, param2, …, paramN)

E.5.3 Event notification functions prototypes

Notification function prototypes are similar to one-way client functions:

dpws_notify_ns__function(struct dpws*, struct wsa_endpoint_ref* event_source,
param1, param2, …, paramN)

The notification function expects a mandatory *event_source* endpoint reference parameter that represents the local service that sends the event.

The additional parameters correspond to the notification function parameters. The corresponding parameters will be absent if an empty structure type has been specified as input parameter.

E.5.4 Building

To generate an event handler executable, you must compile and link your program with the following file:

- *soapHandler.c* (generated by the gSOAP compiler), the skeleton,

or the following file to generate an event source executable:

- *soapServer.c* (generated by the gSOAP compiler), the event stub,

and:

- *soapC.c* (generated by the gSOAP compiler), the marshalling/demarshalling code,
- the DPWS runtime, which includes the gSOAP runtime (note: the gSOAP runtime *stdsoap2.c* must be on the include path):
  - cache.c,
  - config.c,
  - discovery.c,
  - dpws.c,
  - event.c,
  - mdm.c,
  - metadata.c,
  - my_stdsoap2.c,
  - subscmanager.c,
  - *tools.c.*

F. Advanced programming & features

This section deals with advanced topics that may arise when developing real-world applications. It will be completed over time with questions and answers.

F.1 Developing a multi-services application

F.1.1 Introduction

A typical device running a DPWS architecture will feature a few services, be a client to a few others, and may be the destination endpoint of some unsolicited, asynchronous messages. In such a case, it will be necessary to link into the same application the generated code corresponding to several services.

This requirement leads to several problems with the gSOAP/DPWS approach:

- When services share XML Schema definitions, the generated code for those services will contain duplicate structures definitions and marshalling/demarshalling functions, leading to compilation or link errors due to multiple definitions.
- The server should react to several types of messages, namely the requests corresponding to its advertised services, and the asynchronous responses of which it is the destination. This requires to register the dispatch function of each individual service or handler according to the needs.
- As a prerequisite to the above requirement, the server should be able to manage correctly the namespace tables associated to each individual service or handler.

Two approaches can be considered to address the above issues:

- The first solution combines the various service definition files, including those used only as client, into a single, large definition file. This file can then be processed in one pass to generate stubs, skeletons and handlers for all services. Note that this approach is not modular (any change to a given definition file implies the re-generation of the complete application). Another drawback is that the skeleton methods for services which are only invoked by the application are generated anyway, which requires the developer to provide a dummy implementation (which will never be called) for the service function. This approach is described with more details in [gSOAP Guide] § 8.2.11.
- The second approach is based on a modular generation and compilation of the code related to one service, and relies on link edition to build the complete application. Although more appealing, this approach requires some additional care to avoid issues related to multiple definitions of types or symbols. This is the approach that is detailed in the next paragraph.

F.1.2 The library approach for building multi-services applications

gSOAP provides some support for the generation and use of libraries:

- By using appropriate compilation options ('-n' and '-p'), the user can tell the gSOAP compiler to generate code that avoids name clashes at link time by changing files, functions, data and namespace tables name prefixes.
- By using the 'lib' version of generated stub & skeletons that uses macro definitions to remove the redundant source code and makes the marshalling/demarshalling code static, which also avoids code clashes at link time. 'lib' files are bundles of code named *prefixServerLib.c, prefixClientLib.c* or *prefixHandlerLib.c.*

This approach generates some additional requirements:

- Additional common structures, such as header or fault structures, are not provided by the "lib" versions and must be supplied to the linkage step. The DPWS toolkit includes a file named envC.c that includes such definitions (which is the simple result of the gSOAP/DPWS compilation of an empty header file),

- Separate generation of service code means that several namespace mapping tables are generated. This creates two issues:
  i. The DPWS runtime code must be compiled with the WITH_NONAMESPACES directives;
  ii. The appropriate namespace table must be included in the application (with a "#include prefix.nsmap" precompiler directive).

Whatever approach is used, server methods must be registered for every service during the initialization phase, providing all the flexibility needed for a device implementation.

Examples of multi-services applications should be available in the DPWS distribution.

F.2 Reusing definitions of a header file

GSOAP provides an include mechanism that allows to reuse header files and generation. However, this does not prevent to duplicate the marshalling/demarshalling code. If a header file is designed to gather common type definition or even operations, the following steps should be followed:

1. Use the *#module* directive (see [gSOAP Guide] § 9.4) in the file to put in common,
2. Import this module in the other files using the *#import* directive.

F.3 Client, listener & device…

Different types of program may want to use the DPWS toolkit features. The simplest kind is the ***client*** that only invokes remote web services and requires no configuration at all. Besides several servers types (in the sense of a program waiting for an incoming connection) can be considered:

1. ***discovery listener***. This kind of program requires no configuration, just like a client. No service, web service server or handler function is registered but a callback can be used to process *hello* and *bye* WS-Discovery message,
2. ***eventing*** listener. This kind of program requires configuration in order to define the local endpoint. Web service handler function must be registered (but no service because it would become a full device) to process potential incoming events,
3. ***device***. This kind of program has services, server and potentially handler functions registered. It requires a full configuration especially for device metadata defined in [DPWS].

Notes:

1. All server programs use the *dpws_server_init* API that automatically detects, basing on previous configuration operations, which mode is selected,

2. All server programs must use the standard *dpws_accept*/*dpws_serve* API calls scheme,

3. All server programs can also be clients but they shall use separate *dpws* runtime structure for serving and invoking.

F.4 Stopping a device

Stopping a device (which is not addressed in gSOAP) requires the use of two DPWS API functions:

1. *dpws_stop_server()*, to unblock the *dpws_accept()*. This function will return with the error code DPWS_ERR_SERVER_STOPPED and allow the program to exit the main loop,
2. *dpws_shutdown()*, to send the WS-Discovery Bye messages and the WS-Eventing subscription end notifications.

F.5 Metadata Manager (MDM)

The MDM is a built-in (no need to generate or declare it) proprietary (not part of the DPWS specification) web service which allows the user to remotely set the scopes and the friendly names of a DPWS device. It can be invoked using the client API functions: *dpws_set_device_scopes* and *dpws_set_device_friendly_names*. It was originally designed for the SIRENA project.

This service can be disabled using the `DPWS_BOOL_MANAGEMENT_SERVICE` configuration parameter.

F.6 Using DLLs on Windows

In order to avoid recompiling the complete DPWS source code for each application, it is possible to use a precompiled version of the library. The DPWS library is provided in a static (.lib) and dynamic version (.dll). It is completed with a header file (dpws.h), describing the DPWS API.

When using the precompiled version of the library, all services must be generated and used as libraries (see section F.1.2).