CA Gen Integration

Consuming REST Services – C Edition

Christian Kersters

Broadcom Limited
Web: www.broadcom.com
Corporate Headquarters: San Jose, CA

Revision History

Revision	Date	Change Description
V0.9	2020/11/12	Initial version
V1.0	2020/11/17	Revised edition (integration of C. Jamar suggestions)

References

CA Gen Integration Solutions, Christian Kersters, Broadcom, August 2017
https://community.broadcom.com/HigherLogic/System/DownloadDocumentFile.ashx?DocumentFileKey=4a401797-4dfe-4230-a031-273b908e57d3&forceDialog=0

Hello world: getting started with REST APIs
https://www.genivia.com/dev.html#how-rest

gSOAP User Guide
https://www.genivia.com/doc/guide/html/index.html

Richardson Maturity Model
https://restfulapi.net/richardson-maturity-model/

Contents

Revision History 1
References 2
Contents 3
1. Introduction 5
1.1 Richardson Maturity Model 6
1.2 Specification of REST services and data in Level 2-3 APIs 6
1.2.1 REST service identification 6
1.2.1.1 HTTP verb 6
1.2.1.2 MIME type 7
1.2.1.3 URL 7
1.2.2 Exchanging data with Level 2-3 APIs 8
1.2.2.1 Sending data 8
1.2.2.2 Receiving data 8
1.3 Level 3 Web APIs and HATEOAS 8
2 API specifications 9
2.1 OpenAPI specification 9
2.1.1 Data model 9
2.1.2 Services 10
2.2 WADL 10
2.2.1 Data Model 11
2.2.2 Services 12
3 CA Gen integration 13
3.1 Component-Based Development 13
3.1.1 Data Model Specification 13
3.1.2 Functionality Specification 14
3.1.3 Typical definition of a REST operation 15
3.1.4 Implementation 15
3.1.5 Alternative 15
4 Using gSOAP to consume REST Services 16
4.1 Creation of the gSOAP C artifacts 16
4.1.1 WADL pre-processing 16
4.1.2 WADL processing 17

4.1.2.1 WADL specification parsing 17
4.1.2.2 Stub and skeleton compilation 17
4.1.3 API-specific gSOAP documentation 18
4.1.3.1 Doxygen-generated documentation 18
4.1.3.1.1 Data Model documentation 18
4.1.3.1.2 Processes documentation 18
4.2 External Action Block design 20
4.2.1 Include Files 20
4.2.2 Variables declaration 20
4.2.3 Processing 20
4.2.3.1 Communication initialization 21
4.2.3.2 Specification of input data 21
4.2.3.3 Invocation of REST service 21
4.2.3.3.1 URL build 22
4.2.3.3.2 Services without request payload 22
4.2.3.3.3 Services with request payloads 23
4.2.3.4 Error handling 23
4.2.3.5 Fetch of output data 24
4.2.3.6 Communication termination 24
4.3 Library build 24
4.4 Debugging gSOAP communications 24
5 Conclusion 26
Appendix A. Example of External Action Block 27
Appendix B. Windows C Utility Functions 32

1. Introduction

Over the last decade, REST has gained much momentum, compared to the older SOAP protocol, as a solution to exchange messages and integrate workflows among different, independent parties across the Internet.

Reasons for that enthusiasm for REST are multiple, the most important being:

- **Extensive use of the HTTP protocol**, fostering reuse of hardware and software assets across human-based and machine-based consumption (caching, authentication and authorization, …), and making it lightweight
- **Support for multiple data formats**, with most, if not all REST server frameworks supporting both XML and the less verbose JSON formats
- **Flexibility, simplicity and extensibility of APIs**, easing exchange of structured data and code reuse
- **Statelessness**, making it easy, among others, to develop test harness suites.

Due to this success, many solutions have been developed to assist with REST services publication or consumption, or to extend SOAP-based frameworks to also support REST.

Thanks to their reliance on the HTTP protocol, REST services are very easy to consume, even without such specialized framework. Many options are available, in your preferred language, to send HTTP GET, POST, … requests. Also, certainly when the structure of the messages you exchange remains simple, they can easily be created or decoded using a small set of string manipulation and domain conversion functions. When it's not the case, and the structure requires more work, complexity of the task will be significantly reduced by relying on XML or JSON libraries, many of which available as Open Source. *If the services you want to consume are quite independent, in their function or interface[1], this is probably the best approach*.

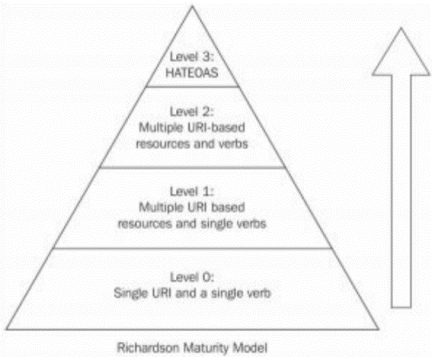
If, on the contrary, the services you want to consume provide a consistent Web API, the REST consumption frameworks provide better alternatives. In addition to give the functions to support all necessary HTTP features, **those solutions, typically, recreate the API data model, based on its formal documentation**. This data model is then populated / queried by your specific consumption logic, using generated functions and fields, and automatically serialized to / de-serialized from the selected message format (typically XML or JSON).

For the C/C++ languages, this is the case with the **gSOAP development toolkit**, well known in the SOAP world, but also supporting REST. Although not as easy as Java-based solutions, gSOAP certainly makes consumption of REST services doable in CA Gen applications implemented in C, even without in-depth knowledge of the language and the HTTP protocol, and it's the objective of this document to guide and help you to do this.

[1] Like SOAP web services converted to REST

Consuming REST Services – C Edition
At Broadcom Mainframe Services for CA Gen, we've extensively used gSOAP to consume SOAP and REST services from CA Gen C external action blocks. Our best practices will be presented here.

1.1 Richardson Maturity Model



Leonard Richardson analyzed a hundred different web service designs and divided them into four categories based on how much they are REST compliant.

In the rest of this document, Levels 0-1 types of REST services will be called ***Ad-Hoc Services***[2], where levels 2-3, thanks to their consistency, truly are ***Web APIs***[3].

While the technical part of this document can apply equally well to levels 1-3, the methodological view of our best practices are much more applicable when consuming Web APIs.

1.2 Specification of REST services and data in Level 2-3 APIs

1.2.1 REST service identification

In Web APIs, target REST services are identified by the following 3 components:

HTTP Verb – URL – MIME Type

1.2.1.1 HTTP verb

The standard HTTP verbs are normally used to specify the type of action the Service provides:

Verb	Meaning
GET	Read
POST	Create
PUT	Update

[2] Level 0 should really be considered as "*XML/JSON Services*", rather than REST, as it's only the content of the message that drives the process
[3] For information, Broadcom Mainframe Services' ***Web API Designer*** Field-Supported Solution can generate Level 1 Ad-hoc Services and Level 2 (and Level 3, using Jboss RESTEasy REST framework) Web APIs

DELETE	[Logical] Delete

HTTP considerations can however influence this clear setup. The most important consideration is that GET requests don't support any message payload (nor should the DELETE ones).

(*At Broadcom Mainframe Services for CA Gen, we avoid as much as possible designs where Read requests require significant / structured input. However, when the impact of this constraint would be too high – like performances or workload – we use POSTs instead, and clearly document the case*).

1.2.1.2 MIME type

There are 2 optional MIME types associated with a request, which are specified in header parameters

- ***Content-Type***, which describes the format of the body that is sent to the service
- ***Accept***, which describes the format of body the consumer expects in the response.

For REST, wherever relevant, both contain ***xml*** or ***json***, and are most generally ***application/xml*** or ***application/json***.

As such, they don't influence the service that is invoked by the HTTP verb and the URL.

Some REST server frameworks, however, make it possible to invoke different services, based on the MIME type. This possibility could be used for service versioning. Such custom MIME types should normally start with "vnd." (for vendor-specific). For instance, vnd.com.broadcom.mf.gen.cse.v01+xml *as Content-Type would mean that we want to access version 1 of a service with a request payload in XML.*

1.2.1.3 URL

In Web APIs, URLs always start with the same base content:

```
http[s]://<host>[:<port>]/<base url>
```

Next part is definition of the target ***resource***[4], with optional ***id*** and ***action***:

```
{/<resource>[/id]}[/action]
```

If no id is specified, the action (or default HTTP verb behavior) applies to the type of resource (like listing for GET or creation for POST), and it applies to the specific resource if the id has been specified (normally defaulting to read for GET, update for PUT or delete for DELETE, as mentioned before).

[4] A resource is similar to an object, containing fields and accessed through a number of methods (which are its REST services)

Based on the identified resource (between <id> and <action>), sub-resources can be accessed, at arbitrary depth (as represented by the “{}”), as long as previous resources have been identified (accompanied by one id).

1.2.2 Exchanging data with Level 2-3 APIs

1.2.2.1 Sending data

In addition to resource identifiers specified in the URL (path parameters), consumers of APIs can also send:

- ***Query parameters*** (*?parm1=xxx&parm2=yyy*)
- ***Header fields*** (***Authorization***, custom fields)
- Request ***payload*** (except for GET/DELETE requests, as mentioned before).

1.2.2.2 Receiving data

REST services will also (normally) send information back to the consumer, as:

- ***HTTP Status code***
- Custom ***Header fields***
- Response ***Payload***.

1.3 Level 3 Web APIs and HATEOAS

HATEOAS stands for ***Hypertext As The Engine Of Application State***. This means that the REST service will send back hyperlinks to the resources it specifies in its response, to ease navigation through the API. Such information makes it very easy for consumers to fetch resource details or related information, based on an initial request.

2 API specifications

As opposed to SOAP, with its unique WSDL specification format, REST supports multiple formal representations, among others:

- ***Swagger***, or its descendant, ***OpenAPI*** (aka Swagger 3) (the most widespread), available in JSON or YAML format
- ***RAML***, YAML-based
- ***WADL***, XML-based, REST equivalent to the WSDL for SOAP, with focus on machine readability.

2.1 OpenAPI specification

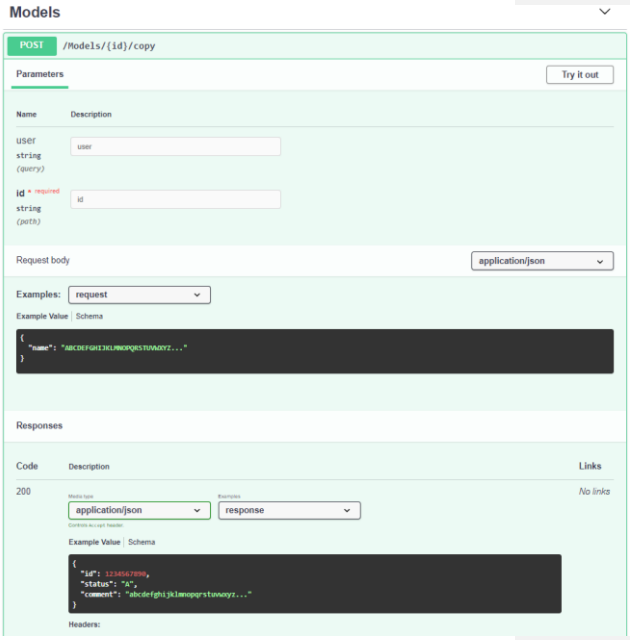
Here are some examples of an OpenAPI specification of a REST API, in native format (JSON in this case) or human representation.

2.1.1 Data model

```
"Model": {
  "type": "object",
  "properties": {
    "ContainsArray": {
      "type": "array",
      "items": {
        "$ref": "#/components/schemas/AggSet"
      }
    },
    "id": {
      "type": "string",
      "description": "
      "
    },
    "name": {
      "type": "string",
      "description": "
      "
    },
    "ownerId": {
      "type": "string",
      "description": "
      "
    },
    "release": {
      "type": "string",
      "description": "
      "
    },
    "lastUpdateDate": {
      "type": "string",
      "description": "
      "
    },
    "lastUpdateTime": {
      "type": "string",
      "description": "
      "
    },
  }
},
```


2.1.2 Services

```
"/Models/{id}/copy": {
  "post": {
   "tags":["Models"],
    "summary": "",
    "description": "",
    "requestBody": {
     "content": {
       "application/json": {
         "schema": {
           "$ref": "#/components/schemas/Model"
         },
         "examples": {
           "request": {
           }
         }
       },
       "application/xml": {
         "schema": {
           "$ref": "#/components/schemas/Model"
         },
         "examples": {
           "request": {
           }
         }
       }
     }
    },
    "responses": {
     "200": {
     },
     "500": {
     }
    },
    "parameters": [
       {
        "name": "user",
        "in": "query",
        "required": false,
        "schema": {
          "type": "string"
        }
       },
   ]
  },
  "parameters": [
     {
      "name": "id",
      "in": "path",
      "required": true,
      "schema": {
        "type": "string"
      }
     },
  ]
},
```



2.2 WADL

The fact that the **WADL** for a REST API has been designed, from the beginning, to be unambiguously processed by machines and its syntax, based on similar conventions as WSDLs, has made it the preferred Web API representation for a number of REST consuming frameworks (and certainly those also supporting SOAP), *among which gSOAP*.

Fortunately, if the REST API you want to consume provides a formal specification in some other representation, you can rely on a number of different services[5] (like APIMATIC - https://www.apimatic.io/transformer/) to convert it to WADL.

There are 2 main possibilities for the design of WADL specifications:

- Put the whole specification into one file
- Separate the grammar of the data model from the rest, in the form of a reference to the xsd of the grammar in the main WADL file.

The second option will be used in this document, as it's easy to map to a data model-based approach to REST API consumption.

2.2.1 Data Model

```
<xs:element name="Model" type="ns0:model"/>
<xs:complexType name="model">
    <xs:sequence>
        <xs:element name="ContainsArray" type="ns0:aggSetArray" minOccurs="0">
        </xs:element>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID">
    </xs:attribute>
    <xs:attribute name="name" type="xs:string">
    </xs:attribute>
    <xs:attribute name="ownerId" type="xs:string">
    </xs:attribute>
    <xs:attribute name="release" type="xs:string">
    </xs:attribute>
    <xs:attribute name="lastUpdateDate" type="xs:string">
    </xs:attribute>
    <xs:attribute name="lastUpdateTime" type="xs:string">
    </xs:attribute>
</xs:complexType>
```

(Note that, in the specific extract of a WADL data model specification above, the *xs:ID* type identifies a [partial[6]] identifier for your specification type. It's not necessarily present in all data model specs).

[5] Broadcom Mainframe Services for CA Gen has also developed a converter from Swagger 2 to WADL, for batch processing
[6] As REST provides support for sub-resources, this id should be combined with the ones of parent resources, if any

2.2.2 Services

```
<wadl:resource path="Models/{id}/copy">
  <wadl:method name="POST">
    <wadl:doc><![CDATA[copy]]></wadl:doc>
    <wadl:request>
      <wadl:representation mediaType="application/json">
      </wadl:representation>
      <wadl:representation mediaType="application/xml" element="ns0:Model">
      </wadl:representation>
      <wadl:param name="user" style="query">
        <wadl:doc><![CDATA[]]></wadl:doc>
      </wadl:param>
    </wadl:request>
    <wadl:response>
      <wadl:doc><![CDATA[logging]]></wadl:doc>
      <wadl:representation mediaType="application/json"/>
      <wadl:representation mediaType="application/xml" element="ns0:Logging"/>
    </wadl:response>
  </wadl:method>
  <wadl:param name="id" style="template">
    <wadl:doc><![CDATA[]]></wadl:doc>
  </wadl:param>
</wadl:resource>
```

3 CA Gen integration

We must here distinguish between the kind of REST Services we want to consume. Are these ***ad-hoc REST services*** or do we want to consume a well-designed ***REST API***?

In the first case, easiest is probably to directly rely on views of existing entities and perform the ad-hoc mapping in an external action block.

In the second case, a much more structured approach will be welcome. As a consequence, this will be the sole focus[7] of this section.

3.1 Component-Based Development

From its specification and our consumer point of view, whatever implementation is behind it, a REST API can be considered as an API to a Microservice. As described in the ***CA Gen Integration Solutions*** White Paper I wrote a few years ago, Microservices map very well to the much older, visionary, concept of Component-Based Development (CBD).

From our perspective, there are 2 major differences with our traditional CBD-based approach:

1. As there is no CA Gen-based component, we must build our specification model from scratch, rather than reusing /adapting pieces of implementation definitions
2. The usual trick CBD practicers use, which links specification definitions with implementation artifacts at runtime, must be replaced by a layer of External Action Blocks (EABs) that will perform data mapping and REST service invocations.

3.1.1 Data Model Specification

```
Subject Area  ┌CSE_SPEC
Entity Type   ├─/SCSE_DMDL
Attribute     │ I /MODEL_ID          (Number, 10, Mandatory, Derived)
Attribute     │  /MODEL_NAME        (Text, 32, Mandatory, Derived)
Attribute     │  /MODEL_OWNER_ID    (Text, 8, Mandatory, Derived)
Attribute     │  /MODEL_RELEASE     (Text, 8, Mandatory, Derived)
Attribute     │  /MODEL_LANG_CODE   (Number, 4, Mandatory, Derived)
Attribute     │  /LAST_UPDATE_DATE  (Date, 8, Mandatory, Derived)
Attribute     │  /LAST_UPDATE_TIME  (Time, 6, Mandatory, Derived)
              │
Entity Type   ├─/SCSE_DOBJ ...
Entity Type   ├─/SCSE_DSETID ...
Entity Type   ├─/SCSE_DUSR ...
Entity Type   ├─/SCSE_ENV ...
Entity Type   ├─/SCSE_LOGGING ...
Entity Type   └─/SCSE_OBJECT_COMPARE ...
```

From the formal specification of the REST API, a specification data model is first constructed. This is a manual operation[8], but the mapping is quite straightforward, from any of the API formal type of representation (refer to the OpenAPI or WADL examples for a confirmation).

As usual for CBD, the owner subject area is of specification type, and the entity types are transient entity, specification or interface type.

[7] Although nothing prevents any of these guidelines to be used in other conditions as well
[8] Although some automation, at least partial, using one of the CA Gen APIs, could easily be developed

3.1.2 Functionality Specification

```
Business Sys          CSE_COMPONENT
Ops Library             ICSE
Action Block              ICSE_COMPARE_MODEL
Action Block              ICSE_COPY_MODEL
Action Block              ICSE_CREATE_AGGREGATE_SET
Action Block              ICSE_DELETE_MODEL
Action Block              ICSE_DELETE_OBJECTS
Action Block              ICSE_DUMMY
Action Block              ICSE_GET_ALL_AGGREGATE_OBJECTS
Action Block              ICSE_GET_MODELS
Action Block              ICSE_GET_MODEL_BY_ID
Action Block              ICSE_GET_OBJECTS_INFO
Action Block              ICSE_MIGRATE_AGGREGATE_SET
```

To mimic what happens with CBD, we take the following approach:

1. Group all the EABs that invoke services of the Web API in their own ***Business System***
2. Create an ***Operations Library*** grouping those action blocks.

With this setup:

a. If consumption if the Web API is needed in multiple models, it's easy and quick to migrate the Operations Library and start consuming its REST services
b. Wherever you want to access the Web API, simply drop the operations library into the runtime environment of the consumer.

Advantage

With this solution, as mentioned above, the whole set of EABs that consume the API will be managed together.

Caveat

There is one small caveat to this approach: by default, CA Gen build processes ignore External Action Blocks in Operations Libraries.

To circumvent this limitation, two easy steps can be taken:

1. In the C build script (in %IEFH%\bt\scripts\build_lm_c.scr:
   a. Identify the following piece of code:

```
{[IF]} EQUAL "{execunit.LMTYPE}" "OPLIB"
   {[FOREACH]} ACBLK AS acblk
      {[IF]} EQUAL "{acblk.OPERATION}" "PUBLIC"
         {[IF]} EQUAL "{acblk.OPLIB}" "{execunit.MEMBER}"
{acblk.MEMBER}
         {[ENDIF]}
      {[ENDIF]}
  {[ENDFOR]}
```

   b. Append, below, the following lines:

```
   {[FOREACH]} EXTERN AS extern
      {[IF]} EQUAL "{extern.OPERATION}" "PUBLIC"
         {[IF]} EQUAL "{extern.OPLIB}" "{execunit.MEMBER}"
{extern.MEMBER}
         {[ENDIF]}
      {[ENDIF]}
  {[ENDFOR]}
```

2. Add some (empty) dummy common action block (here ICSE_DUMMY) to trigger the build process of the Operation Library.

3.1.3 Typical definition of a REST operation

```
ICSE_GET_MODEL_BY_ID of SCSE_DMDL
  IMPORTS:
    Entity View in scse_env (mandatory,transient,import only)
      service_url (mandatory)
    Entity View in scse_dmdl (optional,transient,import only)
      model_id (optional)
  EXPORTS:
    Entity View out scse_logging (transient,export only)
      operation
      comment
    Entity View out scse_dmdl (transient,export only)
      model_id
      model_name
      model_owner_id
      model_release
      last_update_date
      last_update_time
  LOCALS:
  ENTITY ACTIONS:

EXTERNAL
```

As can be seen from the example on the left:

- The EAB, which invokes a service of a REST resource has been declared as ***operation*** of the ***specification type*** that defines the resource in the CA Gen model,
- It receives as imports:
  - The base URL for the Web API
  - The information the service needs as path or query parameter or as payload
- It returns as exports:
  - The information returned by the service as payload, header parameters, …
  - Some execution status information[9]

3.1.4 Implementation

As we use Operation Libraries, which are runtime components (DLLs on Windows or Shared Libraries in Linux / Unix), there is no need to natively build any shared library.

All the EABs that give access to the whole Microservice (or equivalent) will be developed in a single project and grouped together in a static library.

The CA Gen Build process will then convert it to a shared library / DLL.

3.1.5 Alternative

An alternative to this approach is, of course, to avoid customizing the Operation Library build script and directly access the (preferably dynamic) library of consumer EABs.

The drawback of this approach is, however, that the unity of the Web API gets lost at the CA Gen model level.

[9] In this example, the structure for logging-type of payload, returned by some services, has been reused

4 Using gSOAP to consume REST Services

As said in the introduction, there are several ways of consuming REST Services and there is no doubt that gSOAP can be used differently, using features that are provided either in the base product or through complementary plugins.

The approach taken at Broadcom Mainframe Services for CA Gen has been to create an API *data model* on the consumer (gSOAP) side, then to use that data model to interact with gSOAP and the target REST Service.

Also, as our infrastructure is not sensitive to the message format, we've decided to rely on the default XML serialization, rather than use the gSOAP JSON plugin.

4.1 Creation of the gSOAP C artifacts

As briefly told before, this step consists in taking a formal representation of the API (which should normally be provided by its publisher), and processing it using gSOAP tools.

4.1.1 WADL pre-processing

Our experience with gSOAP has shown that some documentation available in WADL specifications prevents the gSOAP REST tools from correctly analyzing the specification.

```
/<wadl:doc / {
    next;
}

/<wadl:doc>/ {
    inDoc = 1;
}

/<\/wadl:doc>/ {
    inDoc = 0;
    next;
}

(inDoc == 1) {
    next;
}

{
    print $0;
}
```

It's consequently strongly advised to remove all pieces of documentation (which are not necessary for machine processing) from the specification. An easy way to do it is with the ***gawk***[10] script, presented on the left side.

(This script copies the whole content of the wadl specification to the output, except for <wadl:doc>…</wadl:doc> sections).

Save the pre-processing script as *removeDoc.awk* and run it with following statement:

gawk –f removeDoc.awk application.wadl application-noDoc.wadl

Commented [CJ1]: added

[10] gawk is a Linux open source tool, which has been ported to many other platforms, like Unix or WIndows

4.1.2 WADL processing

Processing of WADL files by gSOAP is a 2 steps process.

- In the first step, the ***wsdl2h.exe*** utility (parser) is invoked to convert the specification into a gSOAP "header" file,
- In the second step, the ***soapcpp2.exe*** utility (stub and skeleton compiler) will process that header file and provide (in our CA Gen case) standard C files.

Let's look in details at the 2 commands we're normally using.

4.1.2.1 WADL specification parsing

```
wsdl2h.exe -c -R -o application.h -v application-noDoc.wadl -n csens ns0.xsd
```

In this command, we specify, in this order:

- Target of ***c language***
- ***REST specification*** (based on WADL)
- ***Output*** will be a file named ***application.h***
- ***Console logging*** is set to ***Verbose*** (optional)
- ***WADL*** file to be processed is ***application-noDoc.wadl***
- ***Base namespace prefix*** will be ***csens*** (optional, but recommended to avoid conflicts, some short identifier for the consumed Web API)
- ***Secondary input*** is the data model xsd, ***ns0.xsd***.

We strongly recommend to use a non-default (*ns*) namespace, to avoid any runtime conflict between gSOAP-generated artifacts, if your application links to multiple Web APIs.

4.1.2.2 Stub and skeleton compilation

```
soapcpp2.exe -0 -C -c -v application.h
```

Here are specified, again in the order displayed:

- Generation of ***source code*** for ***REST***
- Generation of ***Client-side*** code (as we're consuming services)
- Generation of ***c code***
- ***Console logging*** is set to ***Verbose*** (optional)
- Input file is the **result of the previous step**.

This results in the generation of the following files:

- XML namespace mapping table (***csens2REST.nsmap*** in our case)
- ***soapStub.h*** annotated header file
- ***soapH.h***, main header file
- Serializer for C types (***soapC.c***)
- Client-side stub functions, for invocation of services (***soapClient.c***).

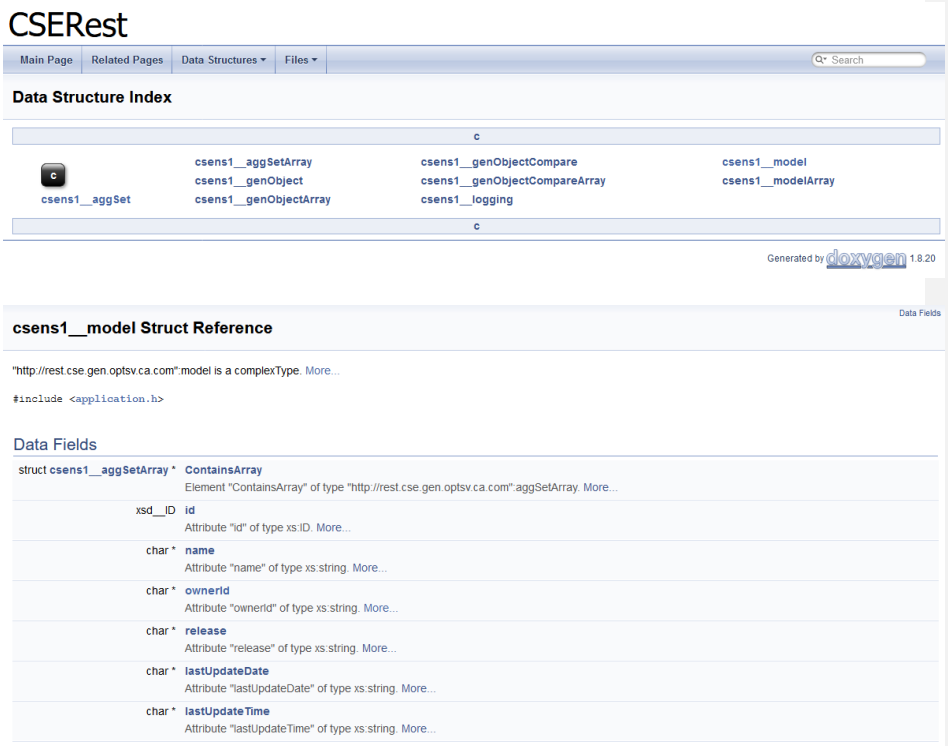
4.1.3 API-specific gSOAP documentation

4.1.3.1 Doxygen-generated documentation

As mentioned in the generated gSOAP header file (in our case, *application.h*), Doxygen (www.doxygen.org) can be used to extract API usage documentation from the header file.

4.1.3.1.1 Data Model documentation

gSOAP header file, processed by Doxygen, is a great tool for the ***Data Model***.



4.1.3.1.2 Processes documentation

The Doxygen documentation, however, does not immediately give any insight into what functions need to be used to invoke the services of the API, and we've to manually analyze the header file to find out the base functions to invoke the services.

The only hint that can be captured from the Doxygen documentation, relative to the functions that invoke the API services, is from its basic processing of the gSOAP header file (*application.h*). Below are 2 examples.

```
/******************************************************************************\
 *                                                                            *
 * Service Operation                                                          *
 *    __csens2__GETResponse                                                   *
 *                                                                            *
\******************************************************************************/

//gsoap csens2 service method-protocol: GETResponse GET
//gsoap csens2 service method-style:    GETResponse document
//gsoap csens2 service method-encoding: GETResponse literal
//gsoap csens2 service method-action:   GETResponse /Models
int __csens2__GETResponse(
    struct csens1__modelArray          *csens1__ModelArray
);
```

This example is about a service to retrieve all models. It's a *GET* operation, as mentioned in (1) and in the method protocol under it, and the base name of this "operation" is *__csens2__GETResponse*.

In the next lines, we also see:

- (2) The default action for this service invocation (we personally never change) is a */Models* URI, which will be appended to the base URL for the Web API
- (3) The structure returned by the REST response is a *ModelArray*.

```
/******************************************************************************\
 *                                                                            *
 * Service Operation                                                          *
 *    __csens2__PUT                                                           *
 *                                                                            *
\******************************************************************************/

//gsoap csens2 service method-protocol: PUT PUT
//gsoap csens2 service method-input-mime-type:  PUT application/x-www-form-urlencoded
//gsoap csens2 service method-encoding: PUT literal
//gsoap csens2 service method-action:   PUT /Models/{id}/deleteObjects
int __csens2__PUT(
    char*                              id,
    char*                              user,
    struct csens1__logging             *csens1__Logging
);
```

In this example, we can see that:

- (1) The operation is a *PUT*, with base name *__csens2__PUT*,
- (2) Its default action URI is */Models/{id}/deleteObjects*, *{id}* representing a variable
- (3) *id* is expected as parameter
- (4) Another parameter is expected, *user*, not part of the service URI, meaning that it's a query parameter
- (5) The structure returned by the REST response is a *Logging* structure.

4.2 External Action Block design

4.2.1 Include Files

```
#include <tiabinc.h>
#include <string.h>
#include "stdsoap2.h"
#include "soapH.h"
#include "csens2REST.nsmap"
```

In addition to the Include directives generated in the EAB stub, the following needs to be added:

- Standard gSOAP header file (***"stdsoap2.h"***)
- Generated main header file (***"soapH.h"***)
- **In one C file only**[11], XML Namespace mappings (***"csens2REST.nsmap"*** or equivalent)
- Also, probably needed, the standard ***<string.h>***.

4.2.2 Variables declaration

In the *User-written code*, the following standard declaration is needed:

```
/*    User-written code should be inserted here    */
   struct soap *soap = soap_new();
```

Other variables will also have to be declared, of standard C types, or types defined in the gSOAP headers. As they are specific to the CA Gen import / export views or consumed REST service, however, they won't be detailed here.

4.2.3 Processing

Processing can be divided into 6 sections:

- Communication initialization, with action and path and query parameters,
- Specification of input data for the REST service (request payload)
- Invocation of REST service
- Error handling
- Fetch of output data of REST service (response payload and optionally response header[12])
- Communication termination.

[11] Including the *nsmap* file into multiple C files will result in multiply-defined symbols at link-edit time
[12] Fetch of response headers is not included in this version of the document)

4.2.3.1 Communication initialization

In its simplest form, communication initialization is limited to some endpoint URL setup, including variables where required.

In our use of gSOAP, we've also encountered the need for basic authentication in our HTTP requests. This is simply implemented by the following:

```
soap->userid = basicUser;
soap->passwd = basicPassword;
```

4.2.3.2 Specification of input data

Specification of path and query parameters has been discussed before (endpoint definition). The only thing that needs to be done is the specification of the request payload, filling in the gSOAP relevant structure.

According to our experience, this is the part where most errors pop up at runtime. Not surprisingly, those errors originate in C memory allocation. Our recommendations are consequently to:

- Always initialize structures, using ***memset*** or ***calloc*** functions, or set all pointers (fields and substructures) to NULL,
- Carefully allocate memory for dynamic variables,
- Free all pieces of memory we've dynamically allocated, in case of failure as in case of success.

Also, depending on the expectations of the REST service, as for the URL formatting, fixed-length attribute views could have to be trimmed before filling in the gSOAP structure.

```
struct csens1__genObjectArray objectsArray;
struct csens1__genObject* objects = NULL;
[...]
// Message setup
objectsArray.__sizeGenObject = a_0173015152_ioa->1003101.1004101.g_in_001ma;
objects = calloc(1, sizeof(struct csens1__genObject) * a_0173015152_ioa->1003101.1004101.g_in_001ma);
objectsArray.GenObject = objects;
for (count = 0; count < a_0173015152_ioa->1003101.1004101.g_in_001ma; count++) {
    objects[count].id = malloc(11);
    sprintf_s(objects[count].id, 11, "%010.0f", a_0173015152_ioa->1003101.1005102.obj_id_004[count]);
}
// Post message
[...]
// Free dynamically allocated memory
for (count = 0; count < a_0173015152_ioa->1003101.1004101.g_in_001ma; count++) {
    free(objects[count].id);
}
free(objectsArray.GenObject);
```

4.2.3.3 Invocation of REST service

gSOAP calls to REST services are rather cryptic. They are declared in ***soapH.h*** and defined in ***soapClient.c***. They are of the form:

```
soap_<verb>_<operation>
```

where :

- ***verb*** is one of call, send, recv
- ***operation*** is the identification of the REST service in the *application.h* gSOAP header file.

As can be seen in the extracts from our gSOAP generated soapClient.c file, ***soap_call*** is equal to ***soap_send*** followed by ***soap_received*** (if send succeeds). Those pieces of code are very insightful, not only to understand what they do, but also to overcome inadequate settings used by gSOAP (as explained later).

4.2.3.3.1 URL build

For the build of URLs, 2 precautions need to be taken, as they could cause the address not to be property recognized or decoded:

1. Fixed-length attribute views (spaces are significant in URLs)
2. Special characters (URLs must be compliant with the syntax defined in RFC 3986).

(Broadcom Mainframe Services has developed/used some utility functions to trim or encode strings according to RFC 3986. See Appendix A for the C code for Windows).

4.2.3.3.2 Services without request payload

- *List of models*:

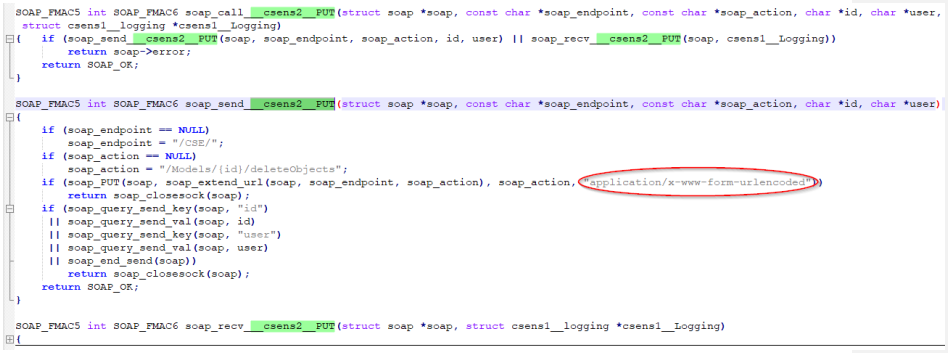
```
SOAP_FMAC5 int SOAP_FMAC6 soap_call___csens2__GETResponse(struct soap *soap, const char *soap_endpoint, const char *soap_action, struct
csens1__modelArray *csens1__ModelArray)
{   if (soap_send___csens2__GETResponse(soap, soap_endpoint, soap_action) || soap_recv___csens2__GETResponse(soap, csens1__ModelArray))
        return soap->error;
    return SOAP_OK;
}

SOAP_FMAC5 int SOAP_FMAC6 soap_send___csens2__GETResponse(struct soap *soap, const char *soap_endpoint, const char *soap_action)
{
    if (soap_endpoint == NULL)
        soap_endpoint = "/CSE/";
    if (soap_action == NULL)
        soap_action = "/Models";
    if (soap_GET(soap, soap_extend_url(soap, soap_endpoint, soap_action), soap_action))
        return soap_closesock(soap);
    return SOAP_OK;
}

SOAP_FMAC5 int SOAP_FMAC6 soap_recv___csens2__GETResponse(struct soap *soap, struct csens1__modelArray *csens1__ModelArray)
{
```

When there is no payload associated with a REST request, the ***soap_call*** function can be used as is:

```
rc = soap_call___csens2__GETResponse(soap, a_0174063831_ioa->1000101.service_url_001, NULL, &modelsArray);
```

4.2.3.3.3 Services with request payloads

- *Objects delete*:



```
SOAP_FMAC5 int SOAP_FMAC6 soap_call___csens2__PUT(struct soap *soap, const char *soap_endpoint, const char *soap_action, char *id, char *user,
 struct csens1__logging *csens1__Logging)
{   if (soap_send___csens2__PUT(soap, soap_endpoint, soap_action, id, user) || soap_recv___csens2__PUT(soap, csens1__Logging))
        return soap->error;
    return SOAP_OK;
}

SOAP_FMAC5 int SOAP_FMAC6 soap_send___csens2__PUT(struct soap *soap, const char *soap_endpoint, const char *soap_action, char *id, char *user)
{
    if (soap_endpoint == NULL)
        soap_endpoint = "/CSE/";
    if (soap_action == NULL)
        soap_action = "/Models/{id}/deleteObjects";
    if (soap_PUT(soap, soap_extend_url(soap, soap_endpoint, soap_action), soap_action, "application/x-www-form-urlencoded"))
        return soap_closesock(soap);
    if (soap_query_send_key(soap, "id")
     || soap_query_send_val(soap, id)
     || soap_query_send_key(soap, "user")
     || soap_query_send_val(soap, user)
     || soap_end_send(soap))
        return soap_closesock(soap);
    return SOAP_OK;
}

SOAP_FMAC5 int SOAP_FMAC6 soap_recv___csens2__PUT(struct soap *soap, struct csens1__logging *csens1__Logging)
{
```

In this example, we see that gSOAP has decided to force a ***Content-Type*** of *application/x-www-form-urlencoded*. This is not what our Web API, nor, probably, what yours expect. As we use gSOAP XML serialization, the type we want to specify is: *application/xml; charset=utf-8*[13].

So, as suggested by the code, we must use the generic *soap_*<***verb***> function (soap_PUT or soap_POST).

To make things easier, instead of writing all those soap_* invocations we see in the implementation of the *soap_send* function, we set up the target endpoint (URL) using normal string functions, then **serialize our XML request payload, including namespace binding**, like this:

```
// /Models/{id}/deleteObjects
sprintf_s(endpoint, sizeof(endpoint), "%s/Models/%010.0f/deleteObjects?user=%s",
    a_0174064508_ioa->1000101.service_url_001, a_0174064511_ioa->1002101.model_id_003, a_0174064513_ioa->1001101.u_user_id_002);

// Message setup
[...]
// Post message
rc = soap_PUT(soap, endpoint, NULL, "application/xml; charset=utf-8")
    || (soap_serialize_csens1__genObjectArray(soap, &objectsArray), 0)
    || soap_put_csens1__genObjectArray(soap, &objectsArray, "csens1:GenObjectArray", "")
    || soap_end_send(soap);
// Receive reply
if (rc == 0) {
    rc = soap_begin_recv(soap) || soap_get_csens1__logging(soap, &logging, NULL, NULL) || soap_end_recv(soap);
}
```

4.2.3.4 Error handling

Error handling can be limited to gSOAP errors in the communication with the server, or extended to include HTTP error status codes.

In the following example, both are taken into account to return some error information to the caller of the EAB:

[13] If our Web API is not using MIME type-based variants / versions

```
if ((rc != SOAP_OK) && (soap->status >= 300)) {
    soap_print_fault(&soap, stdout);
    sprintf_s(a_0174064512_ioa->1006101.status_005, sizeof(a_0174064512_ioa->1006101.status_005), "F");
    globdata->psmgr_exit_msgtype = 'E';
}
else {
    [...]
}
```

(Note that, in this example, only the exit state message type has been specified. A much better solution would be to use a full exit state structure).

4.2.3.5 Fetch of output data

Fetch of output data is quite simple, the only complexity coming from conversion between domains, as shown below.

```
// Decoding of output
a_0173015057_ioa->1002101.1003101.g_out_001ma = modelsArray.__sizeModel;
models = modelsArray.Model;
for (count = 0; count < modelsArray.__sizeModel; count++) {
    id = atof(models[count].id);
    a_0173015057_ioa->1002101.1004102.model_id_003[count] = id;
    strcpy_s(a_0173015057_ioa->1002101.1004102.model_name_003[count], 33, models[count].name);
}
```

4.2.3.6 Communication termination

Finally, the gSOAP structures need to be cleaned up:

```
// Termination
soap_destroy(soap);
soap_end(soap);
soap_free(soap);
```

4.3 Library build

Build of EABs library / archive varies depending on the target environment, and is beyond the scope of this document.

What is needed to build our EAB library:

- Library type: static library (mentioned earlier), as we rely on CA Gen Build Tool to give us a dynamic / shared Operations Library
- C files needed for in the library:
  - External Action Blocks
  - Generated gSOAP C files (soapC.c, soapClient.c)
  - Standard gSOAP C file (stdsoap2.c)

4.4 Debugging gSOAP communications

When compiled with DEBUG macro definition, gSOAP produces 3 files in the folder where the application was started, that can be used for debugging purposes:

- ***SENT.log*** contains messages sent, concatenated
- ***RECV.log*** contains messages received, concatenated
- ***TEST.log*** contains debugging information.

Of course, those files occupy space and give some processing overhead. So, it's recommended not to use them in production.

5 Conclusion

In this document, we've shown (or at least tried to show) that, even without any automation, consuming REST services, even in large, is quite feasible in C.

There is of course much more to the topic than this. We, at Broadcom Mainframe Services, have many more examples, with cases that couldn't be detailed here, and are here to help you.

Appendix A. Example of External Action Block

```
/************************************************************
 *
 *                    Source Code Generated by
 *                           CA Gen 8.6
 *
 *    Copyright (c) 2018 CA Technologies. All rights reserved.
 *
 *    Name: ICSE_DELETE_OBJECTS              Date: 2018/08/27
 *    Target OS:   WINDOWS                   Time: 14:26:59
 *    Target DBMS: <NONE>                    User: kerch01
 *    Access Method: Embedded SQL
 *
 *    Generation options:
 *    Debug trace option not selected
 *    Data modeling constraint enforcement not selected
 *    Optimized import view initialization not selected
 *    High performance view passing selected
 *    Last_statement_num execution selected
 *    Enforce default values with DBMS not selected
 *    Init unspecified optional fields to NULL not selected
 *
 ************************************************************/
/************************************************************
 Data declarations
 ************************************************************/
static char ief_cgen_rlse[] = "CA Gen 8.6";
static char * ss_copyright = "Copyright (c) 2018 CA Technologies. All rights
reserved.";
static char ief_cgen_date[] = "2018/08/27";
static char ief_cgen_time[] = "14:26:59";
static char ief_cgen_ency[] = "9.2.A6";
static char ief_cgen_userid[] = "kerch01";
static char ief_cgen_model[] = "GEN XCIDE";
static char ief_cgen_subset[] = "ALL";
static char ief_cgen_name[] = "ICSE_DELETE_OBJECTS";
#define TGT_EXTERNAL
#define TGT_MULTIVIEW

#include <tiabinc.h>
#include <string.h>
#include "stdsoap2.h"
#include "soapH.h"
#include "utilities.h"
#include "csens2REST.nsmap" // Only needed once

/* * * * * * * * * * * * * * * * * * * * */
/* START OF IMPORT VIEWS */
/* * * * * * * * * * * * * * * * * * * * */
struct w_ioa_0174064508_000
{
struct
{
/* Entity View: IN */
/*        Type: SCSE_ENV */
char service_url_001as;
```

```
char service_url_001[256] /* 255 + 1 */;
} l000l01;
};

static struct w_ioa_0174064508_000  *a_0174064508_ioa;

struct w_ioa_0174064513_001
{
struct
{
/* Entity View: IN */
/*        Type: SCSE_DUSR */
char u_user_id_002as;
char u_user_id_002[9] /* 8 + 1 */;
} l001l01;
};

static struct w_ioa_0174064513_001  *a_0174064513_ioa;

struct w_ioa_0174064511_002
{
struct
{
/* Entity View: IN */
/*        Type: SCSE_DMDL */
char model_id_003as;
double model_id_003;
} l002l01;
};

static struct w_ioa_0174064511_002  *a_0174064511_ioa;

struct w_ioa_0173015152_003
{
struct
{
struct
{
/* Repeating GV:  G_IN */
/*     Repeats: 500 times */
long g_in_001ma;
char g_in_001ac[500];
} l004l01;
struct
{
/* Entity View: IN_G */
/*        Type: SCSE_DOBJ */
char obj_id_004as[500];
double obj_id_004[500];
} l005l02;
} l003l01;
};

static struct w_ioa_0173015152_003  *a_0173015152_ioa;

/* * * * * * * * * * * * * * * * * * * * * */
/* START OF EXPORT VIEWS */
/* * * * * * * * * * * * * * * * * * * * * */
struct w_ioa_0174064512_004
```

```
{
struct
{
/* Entity View: OUT */
/*        Type: SCSE_LOGGING */
char model_id_005as;
double model_id_005;
char operation_005as;
char operation_005[65] /* 64 + 1 */;
char status_005as;
char status_005[2] /* 1 + 1 */;
char comment_005as;
char comment_005[1025] /* 1024 + 1 */;
} l006l01;
};

static struct w_ioa_0174064512_004  *a_0174064512_ioa;

/* * * * * * * * * * * * * * * * * * * * */
/* REPEATING GROUP VIEW STATUS FIELDS  */
/* * * * * * * * * * * * * * * * * * * * */
static char g_in_001fl;
static int g_in_001ps;
static char g_in_001rf;
static int g_in_001mm = 500;
static char long64bit_len[5];
/* * * * * * * * * * * * * * * * * * * * */
/* MISC DECLARATIONS AND PROTOTYPES  */
/*    FOLLOW AS NEEDED:              */
/* * * * * * * * * * * * * * * * * * * * */

static void f_22020211(void);
/*       +->   ICSE_DELETE_OBJECTS               08/27/2018  14:26           */
/*       !       IMPORTS:                                                    */
/*       !         Entity View in scse_env (Transient, Mandatory, Import     */
/*       !                     only)                                         */
/*       !           service_url                                             */
/*       !         Entity View in scse_dusr (Transient, Optional, Import     */
/*       !                     only)                                         */
/*       !           u_user_id                                               */
/*       !         Entity View in scse_dmdl (Transient, Optional, Import     */
/*       !                     only)                                        */
/*       !           model_id                                                */
/*       !         Group View (500) g_in                                     */
/*       !           Entity View in_g scse_dobj (Transient, Optional,        */
/*       !                       Import only)                                */
/*       !             obj_id                                                */
/*       !       EXPORTS:                                                   */
/*       !         Entity View out scse_logging (Transient, Export only)     */
/*       !           model_id                                                */
/*       !           operation                                               */
/*       !           status                                                  */
/*       !           comment                                                 */
/*       !                                                                   */
/*       !     EXTERNAL ACTION BLOCK                                         */
/*       !                                                                   */
/*       +---                                                               */
```

```
/* * * * * * * * * * * * * * * * * * * * */
/* ACTION BLOCK FUNCTION DECLARATIONS  */
/* * * * * * * * * * * * * * * * * * * * */

void ICSEDOBJ(in_runtime_parm1,
in_runtime_parm2,
in_globdata,
im_v_000, im_v_001, im_v_002, im_v_003,
ex_v_000)
char  *in_runtime_parm1;
char  *in_runtime_parm2;
struct ief_globdata  *in_globdata;
struct w_ioa_0174064508_000  *im_v_000;
struct w_ioa_0174064513_001  *im_v_001;
struct w_ioa_0174064511_002  *im_v_002;
struct w_ioa_0173015152_003  *im_v_003;
struct w_ioa_0174064512_004  *ex_v_000;
{
ief_runtime_parm1 = in_runtime_parm1;
ief_runtime_parm2 = in_runtime_parm2;
globdata = in_globdata;
a_0174064508_ioa = im_v_000;
a_0174064513_ioa = im_v_001;
a_0174064511_ioa = im_v_002;
a_0173015152_ioa = im_v_003;
a_0174064512_ioa = ex_v_000;

f_22020211();
return;
}


static void f_22020211(void)
{


/*    User-written code should be inserted here   */
       struct soap *soap = soap_new();
       struct csens1__genObjectArray objectsArray;
       struct csens1__genObject* objects = NULL;
       struct csens1__logging logging;
       char endpoint[256];
       int rc = 0;
       int count;

       strcpy_s(a_0174064512_ioa->l006l01.operation_005, 65, "Objects Delete");
       // /Models/{id}/deleteObjects
       sprintf_s(endpoint, sizeof(endpoint),
              "%s/Models/%010.0f/deleteObjects?user=%s",
              a_0174064508_ioa->l000l01.service_url_001,
              a_0174064511_ioa->l002l01.model_id_003,
              a_0174064513_ioa->l001l01.u_user_id_002);

       // Message setup
       objectsArray.__sizeGenObject = a_0173015152_ioa->l003l01.l004l01.g_in_001ma;
       objects = calloc(1,
              sizeof(struct csens1__genObject) *
               a_0173015152_ioa->l003l01.l004l01.g_in_001ma);
```

```
	objectsArray.GenObject = objects;
	for (count = 0; count < a_0173015152_ioa->l003l01.l004l01.g_in_001ma;
		 count++) {
		objects[count].id = malloc(11);
		sprintf_s(objects[count].id, 11, "%010.0f",
			a_0173015152_ioa->l003l01.l005l02.obj_id_004[count]);
	}
	// Post message
	rc = soap_PUT(soap, endpoint, NULL, "application/xml; charset=utf-8")
		|| (soap_serialize_csens1__genObjectArray(soap, &objectsArray), 0)
		|| soap_put_csens1__genObjectArray(soap, &objectsArray,
			 "csens1:GenObjectArray", "")
		|| soap_end_send(soap);
	// Receive reply
	if (rc == 0) {
		rc = soap_begin_recv(soap)
			|| soap_get_csens1__logging(soap, &logging, NULL, NULL)
			|| soap_end_recv(soap);
	}
	if ((rc != SOAP_OK) && (soap->status >= 300)) {
		soap_print_fault(&soap, stdout);
		sprintf_s(a_0174064512_ioa->l006l01.status_005,
			sizeof(a_0174064512_ioa->l006l01.status_005), "F");
		globdata->psmgr_exit_msgtype = 'E';
	}
	else {
		// Decoding of output
		sprintf_s(a_0174064512_ioa->l006l01.status_005,
			sizeof(a_0174064512_ioa->l006l01.status_005), logging.status);
		sprintf_s(a_0174064512_ioa->l006l01.comment_005,
			sizeof(a_0174064512_ioa->l006l01.comment_005),
			logging.comment);
	}
	// Free dynamically allocated memory
	for (count = 0; count < a_0173015152_ioa->l003l01.l004l01.g_in_001ma;
		 count++) {
		free(objects[count].id);
	}
	free(objectsArray.GenObject);
	// Termination
	soap_destroy(soap);
	soap_end(soap);
	soap_free(soap);
}
```

Appendix B. Windows C Utility Functions

(Those utility functions are available asis, for convenience. There is no guarantee provided by Broadcom for the usage of this code).

```
#include <string.h>
#include <stdio.h>
#include <windows.h>

char rfc3986[256] = { 0 };
int rfcInit = FALSE;

char* trim(char* string, size_t length) {
	size_t trimCount;
	for (trimCount = length - 1; trimCount >= 0; trimCount--) {
		if (string[trimCount] == ' ')
			*(string + trimCount) = '\0';
		else
			break;
	}
	return string;
}

void url_encoder_rfc_tables_init() {
	int i;

	for (i = 0; i < 256; i++) {
		rfc3986[i] = isalnum(i) || i == '~' || i == '-' || i == '.' ||
				i == '_' ? i : 0;
		//           html5[i] = isalnum(i) || i == '*' || i == '-' ||
		//            i == '.' || i == '_' ? i : (i == ' ') ? '+' : 0;
	}
}

char *url_encode(unsigned char *s, char *enc, int maxSize) {
	if (!rfcInit) {
		url_encoder_rfc_tables_init();
		rfcInit = TRUE;
	}
	for (; *s; s++) {
		if (rfc3986[*s]) {
			sprintf_s(enc, maxSize, "%c", rfc3986[*s]);
			enc++;
			maxSize--;
		}
		else {
			sprintf_s(enc, maxSize, "%%%02X", *s);
			enc += 3;
			maxSize -= 3;
		}
	}
	return(enc);
}
```