

# **GT 4.2.1 C WS Core : Developer's Guide**

---

## **GT 4.2.1 C WS Core : Developer's Guide**

### **Introduction**

The C WS-Core developer's guide provides information related to writing and running web services and WSRF-enabled services in C. It includes tutorials walking the developer through creation of services, and clients to interact with services. It includes scenarios for possible configurations that the developer may want. It also provides references to APIs and their documentation.

---

# Table of Contents

1. Before you begin .....	1
1. Feature summary .....	1
2. Tested platforms .....	1
3. Backward compatibility summary .....	1
4. Technology dependencies .....	2
5. Security Considerations for C WS Core .....	2
2. Usage scenarios .....	3
1. Using Wildcards .....	3
2. Using Asynchronous client stubs .....	4
3. Tutorials .....	5
1. Writing Clients for the BlogService .....	5
2. Implementing a Blog Service .....	13
4. Architecture and design overview .....	22
5. APIs .....	23
1. Programming Model Overview .....	23
2. Component API .....	23
I. C WS Core Commands .....	25
globus-wsc-container .....	26
globus-wsrf-cgen .....	28
globus-wsrf-destroy .....	32
globus-wsrf-set-termination-time .....	34
globus-wsrf-query .....	36
globus-wsrf-get-property .....	39
globus-wsrf-get-properties .....	41
globus-wsrf-insert-property .....	43
globus-wsrf-update-property .....	46
globus-wsrf-delete-property .....	49
globus-wsn-get-current-message .....	51
globus-wsn-pause-subscription .....	54
globus-wsn-resume-subscription .....	56
globus-wsn-subscribe .....	58
6. WSDL to C Bindings .....	61
1. Introduction .....	61
2. XML Namespace Mapping .....	61
3. Canonicalization Rules .....	63
4. Types .....	63
5. Client Bindings .....	78
6. Service Bindings .....	86
7. Faults .....	89
8. Errors .....	89
7. GT 4.2.1 Samples for C WS Core .....	91
1. Counter Client .....	91
8. Debugging .....	92
1. Environment variables .....	92
2. Logging .....	92
9. Troubleshooting .....	93
1. C WS Core Errors .....	94
10. Related Documentation .....	96
Glossary .....	97
Index .....	98

---

## List of Figures

6.1. Definition: Namespace to Prefix Mapping Format .....	61
6.2. Mapping table for example schemas .....	62

---

## List of Tables

1. WSRF Core Namespaces and C Prefixes .....	30
2. Common options .....	32
3. Common options .....	34
4. Application-specific options .....	36
5. Common options .....	37
6. Common options .....	39
7. Common options .....	41
8. Common options .....	44
9. Common options .....	47
10. Common options .....	49
11. Application-specific options .....	51
12. Common options .....	52
13. Common options .....	54
14. Common options .....	56
15. Application-specific options .....	58
16. Common options .....	59
6.1. Primitive Type Bindings .....	75
9.1. C WS Core Errors .....	95

---

# Chapter 1. Before you begin

## 1. Feature summary

New Features in the GT 4.2.1 release

- Implementation of the 2004/06 OASIS WS-ServiceGroup working draft specification as an API and a service operation provider. Added support for service-side WSN.
- Command-line tools for accessing WSRF operations (WSN, WSRP, WSRL).

Other Supported Features

- 
- Implementation of the 2004/06 OASIS WSRF and WSN (client) working draft specifications. Implementation of the March 2004 version of the WS-Addressing specification).
- SOAP transport over HTTP/1.1 for clients and services.
- Embeddable service engine with dynamic loading of service code.
- Automatic generation of ANSI-C stubs and service skeletons from Document/Literal WSDL schema and XML schema documents.

Deprecated Features

- None

## 2. Tested platforms

Tested Platforms for C WS Core:

- IA32/Linux/gcc32
- IA64/Linux/gcc64
- x86\_64/Linux/gcc64
- SPARC/Solaris 9/vendorcc32
- PowerPC/AIX 5.2/vendorcc32
- Mac/OS X/gcc32

## 3. Backward compatibility summary

Protocol changes since GT version 4.0.x:

- SOAP messages conform to WSRF schemas instead of previous OGS/OGSA schemas.
- WS-Addressing has been added to the list of supported standards, as defined by the WS-Resource Framework.

- HTTP/1.1 with 'chunked' transfer encoding is used by default.

API changes since GT version 4.0.x:

- The 3.2 cbindings API is obsolete, with no overlap to the new API. Bindings APIs are now generated directly from WSDL.
- The underlying XML/SOAP messaging framework is also new, based on the libxml2 pull parser API.

Schema changes since GT version 4.0.x:

- Schemas are completely new. The WS C Core implements the OASIS WSRF and WSN working drafts specifications (with minor fixes to the 1.2-draft-01 published schemas and with the March 2004 version of the WS-Addressing specification.)

## 4. Technology dependencies

C WS Core depends on the following GT components:

- C Common Libraries
- Non-WS Authentication and Authorization (GSI)
- Globus XIO (used by C WS core for efficient HTTP and TCP transport)

C WS Core depends on the following 3rd party software:

- Libxml2<sup>1</sup> (used by C WS Core for SOAP XML parsing and WSDL parsing)
- OpenSSL<sup>2</sup> (used by C WS Core for Security)
- JavaScript<sup>3</sup> (used by C WS Core as a template language to generate the C bindings from WSDL schemas)

## 5. Security Considerations for C WS Core

### 5.1. Supported Protocols

C WS-Core supports secure transport (https) and secure message (just X509 signing, not encryption).

### 5.2. Secure Transport

With secure transport, the entire container must be run over an https transport. This is done by default for the C container. If the user does not want security in the container, or wants to use secure message instead of secure transport, they should use the `-nosec` argument to `globus-wsc-container`.

For clients, the secure transport is enabled if the contact URI contains the 'https' scheme instead of 'http', so the client doesn't have to enable or disable it explicitly.

---

<sup>1</sup> <http://www.xmlsoft.org/>

<sup>2</sup> <http://www.openssl.org>

<sup>3</sup> <http://www.mozilla.org>

---

# Chapter 2. Usage scenarios

Here we provide some scenarios for using C WS-Core that aren't described in the tutorials.

## 1. Using Wildcards

Both clients and services may need to create or parse instances of `xsd_any` or `xsd_anyType` types. This is necessary when the XML schema defines a type that includes the `xsd_any` or `xsd_anyType` as a type for one of its elements, such as:

```
<xsd:complexType name="TemporalType"> <xsd:sequence>
  <xsd:any minOccurs="1" maxOccurs="1" processContents="lax" />
</xsd:sequence> </xsd:complexType> <xsd:element
name="Temporal" type="tns:TemporalType"/>
```

The content of an instance of `TemporalType` is not restricted by the schema definition, and so must be handled specially at runtime. For serialization and deserialization of wildcard elements, a special global variable of type `globus_xsd_type_info_t` is associated with each type that can be set on the wildcard. For example, if a user wanted an instance of `TemporalType` to contain an instance of an `xsd_dateTime`, the `any` field must be filled in properly. The following bit of C code does this:

```
time_t current; TemporalType temp; xsd_QName * element; xsd_dateTime * time; /* this
is just a struct tm */ ... result = TemporalType_init_contents(&temp); /* check
result */ temp.any.any_info = &xsd_dateTime_info; result =
xsd_dateTime_init(&time); /* check result */ current = time(NULL); /* get the
current time */ result = xsd_dateTime_copy_contents( time, (xsd_dateTime
*)localtime(&current)); /* check result */ temp.any.value = (void *)time; result =
xsd_QName_init(&element); /* check result */ element->Namespace =
globus_libc_strdup("http://temporal.com"); element->local = globus_libc_strdup("Ti
temp.any.element = element; /* now we can serialize it */ result = TemporalType_se
&Temporal_qname, temp, handle, 0); /* check result */
```

This serializes the `TemporalType` to the contain the current timestamp. The resulting serialized elements would look like this:

```
<time:Temporal xmlns:time="http://temporal.com">
<time:Time>Mon Apr 17 10:14:22 CDT 2005</time:Time>
</time:Temporal>
```

If we want to serialize it to a string of the current day of the week, we would do this:

```
time_t current; TemporalType temp;
xsd_QName * element; xsd_string * day; /* this is just a pointer to char * */ ..
TemporalType_init_contents(&temp); /* check result */ temp.any.any_info =
&xsd_string_info; result = xsd_string_init_cstr(&day, "Monday"); /* check
result */ temp.any.value = (void *)day; result = xsd_QName_init(&element); /*
check result */ element->Namespace = globus_libc_strdup("http://temporal.com");
element->local = globus_libc_strdup("Day"); temp.any.element = element; /* now w
serialize it */ result = TemporalType_serialize( &Temporal_qname, temp, handle,
0); /* check result */
```

This allows us to serialize the temporal time element as the day of the week. The resulting serialized elements for this code would look like this:

```
<time:Temporal xmlns:time="http://temporal.com">
  <time:Day>Monday</time:Day> </time:Temporal>
```

So this allows us to inject types into wildcard elements at runtime, and demonstrates how to serialize those wildcards.

For deserialization of wildcard types, a registry is used to lookup the actual type of the element based on QName or the xsi:type attribute. The registry contains key/value pairs of QName to globus\_xsd\_type\_info\_t structures. These structures contain the appropriate information about deserializing the type.

## 2. Using Asynchronous client stubs

A client may wish to perform many invocations of resource property requests to different services (or the same service) at once, without waiting for the response from one request before starting a second request. The asynchronous client stubs generated for each operation allow the client to do this. The example code below shows the implementation of the callback that gets called once the response from a resource property has been received for the CounterService.

```
typedef struct { globus_cond_t cond; globus_mutex_t mutex; } counter_monitor; void
get_rp_counter_value_callback( CounterService_client_handle_t handle, void * user_
globus_result_t result, const wsrp_GetResourcePropertyResponseType *
GetResourcePropertyResponse, CounterPortType_GetResourceProperty_fault_t fault_typ
xsd_any * fault) { counter_monitor_t * monitor = (user_args); xsd_int * rp_value;
if(GetResourcePropertyResponse->any.elements[0].any_info != (&Value_rp_info)) {
/* error - expected Value as the first (and only) resource * property */ } rp_valu
(xsd_int *)GetResourcePropertyResponse->any.elements[0].value;
globus_mutex_lock(&monitor->mutex); monitor->value = *rp_value;
monitor->done = 1; globus_cond_signal(&monitor->cond);
globus_mutex_unlock(&monitor->mutex); } ... counter_monitor_t * monitor; monitor
= globus_malloc(sizeof(counter_monitor_t)); /* check OOM */
globus_cond_init(&monitor->cond, NULL);
globus_mutex_init(&monitor->mutex, NULL); monitor->done = 0; monitor->value
= 0; result = CounterPortType_GetResourceProperty_epr_register( client_handle,
createCounterResponse->EndpointReference, &Value_rp_qname,
get_rp_counter_value_callback, prop_monitor); if(result != GLOBUS_SUCCESS) { ... }
other processing */ globus_mutex_lock(&monitor->mutex); while(!monitor->done)
{ globus_cond_wait(&monitor->cond, &monitor->mutex); /* do other
processing */ } globus_mutex_unlock(&monitor->mutex);
```

This allows us to do other processing while the GetResourceProperty operation is invoked, and the response is returned. For something as simple as the CounterService, the wait for the callback to be called will most likely be short (unless there is network delay). For more complex services, the delay may be longer, and the client may want to perform other processing instead of just waiting.

---

# Chapter 3. Tutorials

## 1. Writing Clients for the BlogService

### 1.1. Introduction: A Blog Service

The Globus Toolkit C WS-Core codebase provides tools and APIs for interacting with web services from a client written in C. It provides additional support for interacting with resource enabled (WSRF) web services. This tutorial provides a walkthrough of the steps to take to create such a C client.

The client we implement interacts with the BlogService, which is a simple example of a Blog web service. See the [Wikipedia entry on Blogs](#)<sup>1</sup> for more information on Blogging. In our simple example, the topic for each Blog is maintained as a *WS-Resource*. The primary ResourceProperty type associated with each Blog resource is an array of strings of all the entries made to that blog topic.

Clients can create new Blog resources with the *createBlogTopic* factory operation, and append their own entries to that resource with the *addEntry* operation. Because the blog stores the entries beyond the lifetime of a single web service invocation (such as *addEntry*), maintaining each blog topic as a resource is a natural use of the framework.

The public interface to a Blog's entry strings is through the *resource property* named `BlogEntry`, and the resource property operations (i.e. *GetResourceProperty*) that are inherited by the BlogService.

The tutorial walks through creation of a blog resource, invoking the *addEntry* operation on that resource, accessing the blog's entries, and finally destroying the blog resource.

For this tutorial we provide the following:

- Complete source for the clients:
  - [create\\_blog.c](#)<sup>2</sup>
  - [add\\_blog\\_entry.c](#)<sup>3</sup>
  - [get\\_blog\\_entries.c](#)<sup>4</sup>
  - [destroy\\_blog.c](#)<sup>5</sup>
- WSDL schemas:
  - [blog.wsdl](#)<sup>6</sup> - Includes the input/output type definitions for the BlogService operations, the ResourceProperty definitions, and the portType definition.
  - [blog\\_bindings.wsdl](#)<sup>7</sup> - Includes the binding definition for the BlogService. The `blog.wsdl` schema file is imported.
  - [blog\\_service.wsdl](#)<sup>8</sup> - Includes the service definition. The `blog_binding.wsdl` schema file is imported.

---

<sup>1</sup> <http://en.wikipedia.org/w/wiki.phtml?title=weblog>

<sup>2</sup> [tutorials/blog/client/create\\_blog.c](#)

<sup>3</sup> [tutorials/blog/client/add\\_blog\\_entry.c](#)

<sup>4</sup> [tutorials/blog/client/get\\_blog\\_entries.c](#)

<sup>5</sup> [tutorials/blog/client/destroy\\_blog.c](#)

<sup>6</sup> [tutorials/blog/blog.wsdl](#)

<sup>7</sup> [tutorials/blog/blog\\_bindings.wsdl](#)

<sup>8</sup> [tutorials/blog/blog\\_service.wsdl](#)

- A GPT package [blog\\_client\\_bindings-0.2.tar.gz](#)<sup>9</sup> of the blog bindings source code. This is the package generated from the WSDL schemas using the `globus-wsrf-cgen` command.
- A tarball [blog\\_client.tar.gz](#)<sup>10</sup> of the counter client source and Makefiles described in this tutorial.

See: [Section 2, “Implementing a Blog Service”](#) for further information on the service side of the implementation. Here, we provide the steps for creating the C client:

## 1.2. Step 0: Acquire a WSDL schema

This is the first step to writing your own client. You must either obtain a pre-existing WSDL schema file (or files), or you must write your own. If you are just going to write a client that interacts with a pre-existing service, the WSDL schema for that service already exists, and you should be able to obtain it from the service author.

For the BlogService, we provide [blog.wsdl](#)<sup>11</sup> that defines the factory operation `createBlogTopic`, the append operation `addEntry`, and the `BlogEntry` resource property for each blog resource. The WSDL schema files should be installed somewhere in the `$GLOBUS_LOCATION/share/schema` tree. In the case of the blog WSDL, you need to install them into `$GLOBUS_LOCATION/share/schema/tutorials/blog/`. This allows the relative paths in the schema import declarations to work.

## 1.3. Step 1: Generate Client Bindings

Once you have the WSDL schema(s) for the service, you need to generate the client bindings from that schema. This will provide the C types and functions (bindings) you need to use to interact with the service. The command used to generate the bindings is `globus-wsrf-cgen`.

To run this command on the blog schema files, they must be placed in `$GLOBUS_LOCATION/share/schema/tutorials/blog/`, as described in Step 0. The command for generating the blog client bindings looks like this:

```
$GLOBUS_LOCATION/bin/globus-wsrf-cgen -no-service -s blog_client \  
-flavor <flavor> -d $PWD/bindings \  
$GLOBUS_LOCATION/share/schema/tutorials/blog/blog_service.wsdl
```

This command will generate the GPT package listed above. The package can be built and installed using the following command:

```
$GLOBUS_LOCATION/sbin/gpt-build bindings/blog_client_bindings-0.2.tar.gz <flavor>
```

## 1.4. Step 2: Write the Client

In order to write a C WS-Core client, the following steps should be followed in general:

### 1.4.1. Include the Client Header

The client bindings generated from [Step 1: Generate Client Bindings](#) include a client header which provides the necessary function declarations to perform the client invocations we need to make. In the case of the BlogService, the [BlogService\\_client.h](#)<sup>12</sup> is the header we need, so it gets included at the top of the file:

---

<sup>9</sup> [tutorials/blog/client/blog\\_client\\_bindings-0.2.tar.gz](#)

<sup>10</sup> [tutorials/blog/client/blog\\_client.tar.gz](#)

<sup>11</sup> [tutorials/blog/blog.wsdl](#)

<sup>12</sup> [tutorials/blog/client/BlogService\\_client.h](#)

```
#include "BlogService_client.h"
```

## 1.4.2. Module Activation

The first step of the client is to activate the module defined for the client. Module activation is a pattern used frequently in the Globus Toolkit. It provides initialization and setup for a particular library, and the libraries it depends on. In this case, the module we are activating is the `BLOGSERVICE_MODULE`, defined in `BlogService_client.h`<sup>13</sup>, as follows:

```
globus_module_activate(BLOGSERVICE_MODULE);
```

## 1.4.3. Client Handle Init

Once the module is activated, the client handle must be initialized:

```
BlogService_client_handle_t  blog_handle;
```

```
...
```

```
result = BlogService_client_init(  
    &blog_handle,  
    NULL, NULL);
```

This handle provides abstraction for messaging and transport configuration parameters, and is used by all Blog service invocations. The second and third parameters are attrs and handler chains that determine how the message is serialized and transported. In this example, we use the default configuration, so the parameters are `NULL`.

In some scenarios, attrs and handlers will need to be setup explicitly by the user.

## 1.4.4. Creating a Resource

Once the client handle is initialized, the next step is to create the blog resource in the `BlogService`. The `create_blog.c`<sup>14</sup> performs resource creation by invoking the `createBlogTopic` factory operation. The bindings call from the client looks like this:

```
createBlogTopicType          createBlogTopic;  
createBlogTopicResponseType * createBlogTopicResponse;  
Blog_createBlogTopic_fault_t create_fault_type;  
xsd_any *                    fault;  
  
...  
  
createBlogTopic.Topic = "Emacs vs. vi: Which is better?";  
createBlogTopic.Creator = "slang";  
  
result = Blog_createBlogTopic(  

```

---

<sup>13</sup> tutorials/blog/client/BlogService\_client.h

<sup>14</sup> tutorials/blog/client/create\_blog.c

```
blog_handle,  
"http://the.service.host:8080/wsrf/services/BlogService",  
&createBlogTopic,  
&createBlogTopicResponse,  
&create_fault_type,  
&fault);
```

This is a code of the `createBlogTopic` invocation, similar to what's in the `create_blog.c`<sup>15</sup> example. The `Blog_createBlogTopic` function is defined in `BlogService_client.h`<sup>16</sup>. The parameters are the initialized blog handle, the endpoint URI to the `BlogService` (i.e. `"http://the.service.host:8080/wsrf/services/BlogService"`), the input and output parameters, and the fault parameters. In this particular example, the `createBlogTopic` input parameter holds the topic for the blog, and the creator of the blog. The `createBlogTopicResponse` output parameter is filled in by the function call, with the `EndpointReference` of the resource created by the `createBlogTopic` invocation. In our example code, we export the `EndpointReference` to a file, which allows us to access it after the `createBlogTopic` process has completed.

```
globus_soap_message_handle_t      epr_out_handle;  
  
...  
  
result = globus_soap_message_handle_init_to_file(  
    &epr_out_handle,  
    "emacs_vi_epr.xml",  
    GLOBUS_XIO_FILE_CREAT);  
  
...  
  
result = wsa_EndpointReferenceType_serialize(  
    &BlogEPR_qname,  
    &createBlogTopicResponse->EndpointReference,  
    epr_out_handle,  
    0);  
  
...  
  
globus_soap_message_handle_destroy(epr_out_handle);
```

Now we must destroy the response from `createBlogTopic` invocation:

```
createBlogTopicResponse_destroy(createBlogTopicResponse);
```

### 1.4.5. Invoking a Resource Operation

Once the `EndpointReference` has been written to file, we have a reference to the blog resource, so we can call the `addEntry` operation on that resource from another process. This is what the `add_blog_entry.c`<sup>17</sup> client example does. The `EndpointReference` for the blog resource is first imported from the file:

```
globus_soap_message_handle_t      epr_in_handle;
```

---

<sup>15</sup> `tutorials/blog/client/create_blog.c`

<sup>16</sup> `tutorials/blog/client/BlogService_client.h`

<sup>17</sup> `tutorials/blog/client/add_blog_entry.c`

```
...  
  
    result = globus_soap_message_handle_init_from_file(  
        &epr_in_handle,  
        "emacs_vi_epr.xml");  
  
...  
  
    result = wsa_EndpointReferenceType_init(&blog_resource_reference);  
  
...  
  
    result = wsa_EndpointReferenceType_deserialize(  
        &BlogEPR_qname,  
        blog_resource_reference,  
        epr_in_handle,  
        0);  
  
...  
  
    globus_soap_message_handle_destroy(epr_in_handle);
```

Once the EndpointReference is imported, the addEntry operation is invoked as follows:

```
addEntryType                entry;  
addEntryResponseType *      blog_entries;  
  
Blog_addEntry_fault_t      add_fault_type;  
xsd_any *                  fault;  
  
entry.Comment = "What's vi??";  
entry.Author = "EmacsPowerUser";  
  
result = Blog_addEntry_epr(  
    blog_handle,  
    blog_resource_reference,  
    &entry,  
    &blog_entries,  
    &add_fault_type,  
    &fault);
```

For this invocation, we're using the `Blog_addEntry_epr` function (instead of `Blog_addEntry`). This allows us to pass in the EndpointReference of the resource directly as the second parameter (that's why the function ends in `_epr`). The first parameter is the client handle, The third and fourth parameters are the input and output parameters to the operation (the blog entry to add, and the resulting entries on the blog), followed by the fault parameters. Once this function call returns successfully, the `addEntryResponse` parameter will contain all the entries made to the blog. This call can be made subsequently and entries will continue to be appended to the resource. Once the response is no longer needed after a call to `Blog_addEntry_epr`, we must destroy it:

```
xsd_string_destroy(addEntryResponse);
```

The output of running `add-blog-entry` will look something like this:

```
./add-blog-entry emacs_vi_blog.xml "Emacs rocks!" anonymous
```

```
BLOG ENTRIES:
```

```
On Wed Dec 22 04:57:42 CST 2004, anonymous said: "Emacs rocks!"
```

```
On Tue Oct 26 01:01:11 CST 2004, wq said: "CTRL-ALT-SHIFT-X CTRL-C...I'm running out of fi
```

```
On Thu Aug 12 10:44:32 CST 2004, EmacsPowerUser said: "What's vi??"
```

## 1.4.6. Getting a Resource Property Value

The WSDL schema for the `BlogService` defines a Resource Property `BlogEntry` as part of the resource property document for the `Blog` port type. This resource property allows us to access the state of the resource (get the entries) with the `GetResourceProperty` operation defined in the `WS-ResourceProperties` schema and inherited by the `Blog` portType. The `get_blog_entries.c`<sup>18</sup> client example performs this operation on the `Blog` resource. The invocation is made as follows:

```
#include "BlogEntry.h"

...

wsrp_GetResourcePropertyResponseType *      RPResponse;
Blog_GetResourceProperty_fault_t          getrp_fault_type;
xsd_any *                                  fault;

...

result = Blog_GetResourceProperty_epr(
    blog_handle,
    blog_resource_reference,
    &BlogEntry_qname,
    &RPResponse,
    &getrp_fault_type,
    &fault);
```

In this function call, the client handle and endpoint reference are passed as the first two parameters. The third parameter (the operation input) is the qualified name of the Resource Property. In this case, the QName is declared in the generated header `BlogEntry.h` as the global variable `BlogEntry_qname`. The output parameter `RPResponse` is the response from the `GetResourceProperty` operation. On successful completion of the function, this response parameter will contain the value(s) of the ResourceProperty. Because resource properties can have any type, the response is deserialized as an array of `xsd_any *` instances. In order to access the actual value from this structure, the type of the `xsd_any *` instance must be verified to match the expected type:

```
if(RPResponse->any.elements[i].any_info->type !=
    (&BlogEntry_qname) &&
    (RPResponse->any.elements[i].any_info->type !=
```

---

<sup>18</sup> `tutorials/blog/client/get_blog_entries.c`

```
    (&Blog_BlogEntry_rp_qname))
  {
    /* error! Unexpected type */
  }
}
```

What's happening here? The `wsrp_GetResourcePropertyResponseType` structure contains the field `any`, which is an `xsd_any_array`. This array is assumed to contain one element at index 0. In order to check that the element was deserialized as the appropriate element (i.e. `BlogEntry`), we must compare the `any_info` field against the reference to the global variable `BlogEntry_qname` declared in `BlogEntry.h`.

Once the type of the element in the response is verified, we can access the value contained in the `value` field of the `xsd_any`.

```
blog_entry = *RPResponse->any.elements[i].value;

printf("BLOG ENTRIES:\n\n%s\n", blog_comments);
```

After the value of the resource property has been accessed, we need to destroy the response instance created by the `Blog_GetResourceProperty_epr` function call:

```
wsrp_GetResourcePropertyResponseType_destroy(RPResponse);
```

The output of running `get-blog-entries` will look something like this:

```
./get-blog-entries emacs_vi_blog.xml
```

```
BLOG ENTRIES:
```

```
On Wed Dec 22 04:57:42 CST 2004, anonymous said: "Emacs rocks!"
```

```
On Tue Oct 26 01:01:11 CST 2004, wq said: "CTRL-ALT-SHIFT-X CTRL-C...I'm running out of fi
```

```
On Thu Aug 12 10:44:32 CST 2004, EmacsPowerUser said: "What's vi??"
```

## 1.4.7. Destroy the Resource

In order to destroy the resource we've created after all our invocations to it are complete, we use the `Destroy` operation defined in `WS-ResourceLifetime` schema and inherited by the `Blog portType`. The `destroy_blog.c`<sup>19</sup> client is an example of using this operation for the `blog` resource. The example imports the resource reference, calls the `Destroy` operation, and then removes the file that referenced the resource.

```
wsrl_DestroyType           Destroy;
wsrl_DestroyResponseType * DestroyResponse;
Blog_Destroy_fault_t      destroy_fault_type;
xsd_any *                  fault;

result = globus_wsrf_core_import_endpoint_reference(
    "emacs_vi_blog.xml", &blog_resource_reference, NULL);
```

---

<sup>19</sup> `tutorials/blog/client/destroy_blog.c`

```
...  
  
result = Blog_Destroy_epr(  
    blog_handle,  
    blog_resource_reference,  
    &Destroy,  
    &DestroyResponse,  
    &destroy_fault_type,  
    &fault);
```

As with the previous `EndpointReference` invocations, the first two parameters passed to this function are the client handle and the endpoint reference to the resource. In the case of invoking the `Destroy` operation, the `Destroy` and `DestroyResponse` input and output parameters are just empty structures and don't contain any pertinent information. Nevertheless, the `DestroyResponse` variable should be cleaned up after the `Destroy` operation has completed:

```
wsr1_DestroyResponse_destroy(DestroyResponse);
```

## 1.4.8. Cleanup

Once all the desired invocations have completed for a particular process, the client handle needs to be destroyed, and the module must be deactivated.

```
Blog_client_handle_destroy(blog_handle);  
  
globus_module_deactivate(BLOGSERVICE_MODULE);
```

These calls exist in each of the client examples.

## 1.5. Step 3: Build the Client

Now you've written an end-to-end C WS-Core WSRF-enabled client. In order to compile the client we demonstrate how to write a Makefile for it. First, the following command must be run:

```
$GLOBUS_LOCATION/bin/globus-makefile-header \  
  --flavor=<flavor> <package> \  
> MyMakefile.include
```

Assuming you compiled the Globus Toolkit with a `gcc32dbg` flavor, and using the `blog` client bindings package from this tutorial, the command would be:

```
$GLOBUS_LOCATION/bin/globus-makefile-header \  
  --flavor=gcc32dbg blog_client_bindings \  
> BlogClientMakefile.include
```

The resulting `BlogClientMakefile.include`<sup>20</sup> contains include and olink definitions for our client. Now we just need to write a Makefile, using the variables defined in the output of the `globus-makefile-header` command. We've provided a `blog` client `Makefile`<sup>21</sup>. Once your Makefile is written, running `make` will generate the client executables. At this point you're not quite ready to run it. The client needs to have a service running somewhere to interact

---

<sup>20</sup> `tutorials/blog/client/BlogClientMakefile.include`

<sup>21</sup> `tutorials/blog/client/Makefile.example`

with. See [Section 2, “Implementing a Blog Service”](#) in order to create and run a BlogService that you can invoke with your new client.

## 2. Implementing a Blog Service

### 2.1. Introduction: A Blog Service

The Globus Toolkit's C WS-Core component provides tools and APIs for creating web services in C. It also provides additional support for creating web services which are WSRF-enabled, meaning the service can manage resources and the associated resource properties. This tutorial provides a walkthrough of the steps needed to create a WSRF-enabled service in C, from defining a WSDL schema for the service to actually running the service in the C service container.

The service we implement in this tutorial is the BlogService, which is a simple service that allows new Blog topics to be created as resources, and then allows comments to be added to a particular Blog topic. See the [Blog Wikipedia entry](#)<sup>22</sup> for more information on Blogs.

In our BlogService, the primary resource property is the BlogEntry element, which is an array of BlogEntryType instances containing the comment, author, and timestamp of each entry posted to the Blog topic. For the tutorial, we will demonstrate how to generate the service stubs and skeletons for the BlogService, and how to provide the service implementation, including creation of new Blog topics as resources, and adding new blog entries to the BlogEntry Resource Property. For the purposes of this tutorial, we provide the following:

- WSDL schema files for the BlogService:
  - [blog.wsdl](#)<sup>23</sup> - Includes the input/output type definitions for the BlogService operations, the ResourceProperty definitions, and the portType definition.
  - [blog\\_bindings.wsdl](#)<sup>24</sup> - Includes the binding definition for the BlogService. The blog.wsdl schema file is imported.
  - [blog\\_service.wsdl](#)<sup>25</sup> - Includes the service definition. The blog\_binding.wsdl schema file is imported.
- Source file for the complete BlogService implementation:
  - [BlogService\\_skeleton.c](#)<sup>26</sup>
- A GPT package [blog\\_service\\_bindings-0.2.tar.gz](#)<sup>27</sup> that contains the complete BlogService implementation (includes the skeleton from the above bullet).

This tutorial defines 5 steps needed to create any WSRF-enabled service using C WS-Core, and then provides example walkthroughs of those steps with the BlogService.

### 2.2. Step 1: Acquiring a WSDL Schema

You must either obtain pre-existing WSDL schema files or write your own. The schema files must contain a service definition that defines the service. Please note that the C WS-Core only supports document/literal style WSDL schema files at present.

---

<sup>22</sup> <http://en.wikipedia.org/w/wiki.phtml?title=Blog&redirect=no>

<sup>23</sup> [tutorials/blog/blog.wsdl](#)

<sup>24</sup> [tutorials/blog/blog\\_bindings.wsdl](#)

<sup>25</sup> [tutorials/blog/blog\\_service.wsdl](#)

<sup>26</sup> [tutorials/blog/service/BlogService\\_skeleton.c](#)

<sup>27</sup> [tutorials/blog/service/blog\\_service\\_bindings-0.2.tar.gz](#)

For the BlogService, we provide `blog.wsdl`<sup>28</sup> that defines the factory operation `createBlogTopic` and the append operation `addEntry`, as well as the `BlogEntry` resource property for each blog resource.

## 2.3. Step 2: Generating Service Bindings

Once you have the WSDL schema(s) for the service, you need to generate the service bindings from that schema. This will provide the C skeleton functions for the service implementation. The command used to generate the bindings is `globus-wsrf-cgen`.

To run this command on the Blog schema files, they must be placed in `$GLOBUS_LOCATION/share/schema/tutorials/blog/`, so that the relative import paths are correct. The command for generating the blog service bindings looks like this:

```
$GLOBUS_LOCATION/bin/globus-wsrf-cgen -no-client -s blog_service \  
-d $PWD/bindings -flavor <flavor> \  
$GLOBUS_LOCATION/share/schema/tutorials/blog/blog_service.wsdl
```

This command generates source and header files for the service, and as a final step, creates a GPT package (a `.tar.gz` file) that contains all the source, headers and necessary build files. Building this package is described in [Section 2.5, “Step 4: Building/Installing the Service Package”](#). The above command generates build files and type bindings files in the bindings directory as a sub-directory of the current directory. Service specific files are output to a sub-directory of bindings named `<service name>` (`$PWD/bindings/<servicename>/`). In this example the sub-directory is named `BlogService`.

The `-d <dir>` argument outputs the generated files to `<dir>`. Use the `-help` argument to get further info.

## 2.4. Step 3: Writing the Service implementation

Once the service binding generation has completed, the service skeleton functions will reside in the `<service name>_skeleton.c` source file contained in the `<service name>` directory. This is the file with the operation functions that must be filled in to complete the implementation of the service. For this example, the file we must modify is `BlogService/BlogService_skeleton.c`. This source file includes skeleton functions for each of the operations defined in the `blog.wsdl`<sup>29</sup> schema file. The two operations that need to be implemented are `createBlogTopic` and `addEntry`. The associated functions in `BlogService_skeleton.c`<sup>30</sup> are `Blog_createBlogTopic_impl` and `Blog_addEntry_impl`.

### 2.4.1. Creating a Resource

In the WS-ResourceFramework, operations which create new resources and provide us with references to them are called *factories*. In the BlogService, the `createBlogTopic` operation is the factory that creates a new resource (a new Blog topic), and returns a reference to it (as an `EndpointReference`). This function creates the resource instance, fills in the `EndpointReference` to be returned, and creates a resource property `BlogEntry` on the resource.

### 2.4.2. The Resource ID

As the first step of creating a resource in our `Blog_createBlogTopic_impl` function, we must acquire a resource ID. The resource ID is an application specific object that acts as a unique identifier for the resource within the service, and gets embedded within the `EndpointReference` for the new resource. For C WS-Core, the resource ID must be in the

---

<sup>28</sup> `tutorials/blog/blog.wsdl`

<sup>29</sup> `tutorials/blog/blog.wsdl`

<sup>30</sup> `tutorials/blog/service/BlogService_skeleton.c`

form of a string. In many services, the resource ID is a UUID string, generated by the `globus_uuid_create` function. See the UUID library documentation for further info.

In the case of the `BlogService`, we assume that no two Blogs created by the same person will have the same topic, so we can join the author and topic strings together as the resource ID for the new resource we are about to create.

```
globus_result_t
Blog_createBlogTopic_impl(
    globus_service_engine_t      engine,
    globus_soap_message_handle_t message,
    globus_service_descriptor_t * descriptor,
    createBlogTopicType * createBlogTopic,
    createBlogTopicResponseType * createBlogTopicResponse,
    const char ** fault_name,
    void ** fault)
{
    char * resource_id;
    globus_result_t result = GLOBUS_SUCCESS;

    GlobusFuncName(Blog_createBlogTopic_impl);
    BlogServiceDebugEnter();

    blog_id = globus_common_create_string(
        "%s#%s", createBlogTopic->Creator, createBlogTopic->Topic);
```

The `blog_id` is then passed to the `globus_resource_create` function, which will create a managed resource and return it in `blog_resource`.

```
result = globus_resource_create(
    blog_id,
    &blog_resource);

...

result = BlogServiceInitResource(blog_id);
```

The second call in the code listing above is the service's resource init function, which allows the operation providers to initialize the resource properties of the resource you've just created. For example, the `WS-ResourceLifetime` operation provider adds `CurrentTime` and `TerminationTime` resource properties to the resource.

The bindings for any service definition will include a `<service name>InitResource([resource id]);` macro which calls the resource initialization functions for each operation provider the service includes.

### 2.4.3. The EndpointReference (EPR)

Once the resource is created the `EndpointReference` must be created. The first step is to initialize a reference property of the EPR, which will contain the resource ID we just created. The reference property is a field in the `wsa_EndpointReferenceType` type. Since the property can be anything, it is typed to the XSD wildcard `xsd_any`, which we must create an instance of and initialize to contain the appropriate type and value for the reference property.

```
result = xsd_any_init(&reference_property);
reference_property->any_info = &xsd_string_info;
```

```
...

result = xsd_QName_init(reference_property->element);

...

reference_property->element->Namespace = globus_libc_strdup(
    BlogService_service_qname.Namespace);
reference_property->element->local = globus_libc_strdup("BlogID");

result = xsd_string_copy_cstr(
    (xsd_string **)&reference_property->value,
    blog_id);
```

The `xsd_any` type we initialize has 3 important fields. The `any_info` field contains the type information used by the marshalling engine to determine how to serialize the reference property. In this case the reference property is just a string, so we set the `any_info` field to the globally defined `xsd_string_info` variable. For more information on using wildcards in your service implementation, see [Chapter 2, Usage scenarios](#).

The `element` field in `xsd_any` is a *QName* of the element to define for serializing the type. In the `BlogService` case, we set the element to `http://globus.org/blog#BlogID`. The other field we need to set in the reference property is the `value`, which is a `(void *)`, set to the pointer of the instance of the resource id (in this case the blog id string). We use the `xsd_string_copy_cstr` function to actually copy the contents of the string to the `value` field.

Once the reference property has been initialized, we can create the `EndpointReference`. The `globus_wsrf_core_create_endpoint_reference` convenience function has been provided to create the endpoint reference.

```
result = globus_wsrf_core_create_endpoint_reference(
    engine,
    BLOGSERVICE_BASE_PATH,
    &reference_property,
    &createBlogTopicResponse->EndpointReference);
```

This call takes the `engine` passed into the skeleton function, the base path of the URI for the service (each service has a `<service name>_BASE_PATH` variable defined), and the reference property we just initialized. The resulting `EndpointReference` must be set to the `EndpointReference` field in the `createBlogTopicResponse` variable passed into the skeleton function.

## 2.4.4. The Resource Property

As the last step of the `Blog_createBlogTopic_impl` function, we set the `BlogEntry` resource property of the resource. Since the `Blog` initially doesn't contain any entries, we set the resource property to an empty array. We will add new entries to this resource property in the `Blog_addEntry_impl` skeleton function.

```
result = BlogEntryType_array_init(&blog_entries);

...

result = globus_resource_create_property(
    blog_resource,
    &Blog_BlogEntry_rp_qname,
```

```
&BlogEntry_array_info,  
blog_entries);
```

The arguments passed to this function are the created resource, the QName of the resource property (in this case, *BlogEntry*), the info variable of the resource property type to create, and the empty blog array instance. See the [Resource API](#)<sup>31</sup> for further documentation.

### 2.4.5. Add an Entry to the Blog Topic

Once a resource has been created, clients will invoke the `addEntry` operation to add new entries to the blog. The implementation of the `Blog_addEntry_impl` adds the new entry to the blog topic.

### 2.4.6. Access the Resource

The resource is accessed through the `EndpointReference` contained in the message. The utility function `globus_wsrf_core_get_resource` is used to access the resource. The `EndpointReference` is accessed through the first parameter (`message`) passed to the function.

```
result = globus_wsrf_core_get_resource(  
    message,  
    descriptor,  
    &blog_resource);
```

Information about how the resource ID is accessed from the `EndpointReference` is maintained by the service descriptor, so this gets passed in as the second parameter (`service`).

### 2.4.7. Get the Resource Property

Once we have the resource we can access the `BlogEntry` resource property using the `globus_resource_get_property` function.

```
result = globus_resource_get_property(  
    resource,  
    &Blog_BlogEntry_rp_qname,  
    (void **)&blog_entries,  
    NULL);
```

The first parameter is the blog resource we just accessed, the second parameter is the QName of the `BlogEntry` resource property. `Blog_BlogEntry_rp_qname` is a global variable declared in `BlogService.h`. Global QName variables exist for each resource property in a service. The third parameter is the array of blog entries we want to get. The last parameter is the type info structure of the resource property we're accessing. Since we know the type of the resource property, we can just set this to `NULL`.

### 2.4.8. Add the Blog Entry

Now that we have the array of blog entries, we need to add a new element to the end of it with the values of the entry. Each array type generated from an XML schema document has an associated `_array_push` function, which creates a new instance of the type and adds it to the end of the array, returning the new instance. In this case, we create a new entry at the end of the array with the `BlogEntryType_array_push` function.

---

<sup>31</sup> [http://www.globus.org/api/c-globus-4.2.1/globus\\_c\\_wsrf\\_resource/html/index.html](http://www.globus.org/api/c-globus-4.2.1/globus_c_wsrf_resource/html/index.html)

```
new_entry = BlogEntryType_array_push(blog_entries);
```

Now we need to fill in this entry with the values passed into the skeleton function.

```
tstamp = time(NULL)
result = xsd_dateTime_copy_contents(
    &new_entry->Timestamp,
    (xsd_dateTime *)localtime(&tstamp));

...

result = xsd_string_copy_contents(
    &new_entry->Author,
    (xsd_string *)&addEntry->Author);

...

result = xsd_string_copy_contents(
    &new_entry->Comment,
    (xsd_string *)&addEntry->Comment);
```

These functions copy the entry's comment and author from the input parameter to the new entry instance we've created. The timestamp of the entry is set to the current local time. This completes the addition of a resource property value to the resource property maintained by the resource instance.

The `addEntry` operation expects as the response a list of the entries in the Blog. Since this is just the array of blog entries that we just added to, we can simply copy this array to the response output parameter:

```
result = BlogEntryType_array_copy_contents(
    &addEntryResponse->BlogEntries,
    blog_entries);
```

## 2.4.9. Resource Finish

As a last step of the `Blog_addEntry_impl` function, we need to release the blog resource we accessed in the first step. This allows the resource management computeroutput to handle locking and reference counting for the resource.

```
globus_resource_finish(blog_resource);
```

## 2.4.10. Other Issues

In this section we describe other parts of implementing the skeleton functions that might be of interest.

## 2.4.11. Service Initialization

Besides the skeleton functions defined for each operation in a service, `BlogService_skeleton.c` also contains functions for initializing and finalizing the BlogService. The `BlogService_init` function should contain any service specific computeroutput that needs to be run when the service is loaded, and the `BlogService_finalize` function should contain computeroutput that needs to be run when the service is unloaded (presumably cleanup from `BlogService_init`). These functions most likely can remain empty no-ops, but if for example you want a service

to have persistent resources which exist throughout the lifetime of the service, they should be created in the service's `init` function and destroyed in the `finalize` function.

## 2.4.12. Error Handling

Almost all of the function calls in our `BlogService` return a `globus_result_t` type. The `globus_result_t` informs the caller of the success or failure of the function call, and is used to reference the error object created if an the function call failed. The standard practice in the Globus Toolkit for handling errors is to check the return value of the function:

```
if(result != GLOBUS_SUCCESS)
```

and if an error occurred, either chain the error or handle the error at that level (exit the process, print an error message, etc.). The skeleton functions we've implemented in this tutorial have a `globus_result_t` return value, so the skeleton function needs to create and return error values if and when they occur within the service implementation. The bindings generated for a service include macros for each operation in the service's header file that create `globus_result_t` error values to be returned by the skeleton function. For example, the signatures of the macros generated for the `addEntry` operation are:

```
globus_result_t  
Blog_addEntry_error(const char *);
```

```
globus_result_t  
Blog_addEntry_chain_error(globus_result_t, const char *);
```

In general, each operation will have an associated error create function that takes a string and returns a `globus_result_t` error as well as an error function that takes a base error `globus_result_t` and a string and returns a new `globus_result_t`.

The first function macro listed is useful for error cases where the error is the primary base cause, while the second function is useful when another globus function has been called and value which is not equal to `GLOBUS_SUCCESS`.

## 2.4.13. Operation Providers

For the operations inherited from the WSRF schemas (*GetResourceProperty*, *Destroy*, *SetTerminationTime*), their implementation has already been provided for us. This is achieved using operation providers, which replace the functions defined in the `BlogService_skeleton.c` source file with generic pre-defined versions of those functions when the service is loaded by the container. Even though the contents of those functions remain empty in the skeleton source file, they don't get used, so they can be safely ignored.

## 2.4.14. Service-Side Notifications

`BlogService_skeleton.c` also includes functions for the *Subscribe* and *GetCurrentMessage* operations that are part of the *WS-BaseN* schema (inherited by the `BlogService`), but the C WS-Core currently doesn't provide implementations of *NotificationProducer* or *SubscriptionManager* at present, so these skeleton functions can remain empty as well.

## 2.5. Step 4: Building/Installing the Service Package

### 2.5.1. Packaging

Once the service implementation is complete, the service package can be re-packaged (create the tarball) with the implemented computeroutput using `make`. Change the working directory to the directory the bindings were generated in, and run:

```
make dist
```

This will create (or re-create) the `blog_service_bindings-0.2.tar.gz` package in that directory with the new service implementation. This package can be distributed to any machine with a CWS-Core installation and installed there.

### 2.5.2. Building

To build the package you just created, run the following command:

```
$GPT_LOCATION/sbin/gpt-build blog_service_bindings-0.2.tar.gz <flavor>
```

This will compile the source files for the types and service and build them into a library module named `libblog_service_bindings.so` (the suffix of the library may differ depending on the platform). The header files are installed into `$GLOBUS_LOCATION/include/<flavor>` and the library is installed in `$GLOBUS_LOCATION/lib/<service base path>`.

## 2.6. Step 5: Running the Service Container

Once the BlogService library module has been installed, the service container can be run and the BlogService can be invoked, causing execution of the service implementation. The service container is run with the command:

```
$GLOBUS_LOCATION/bin/globus-wsc-container
```

## 2.7. Step 6: Debugging the Service Implementation

### 2.7.1. Adding Debug Statements

Each service module includes debugging macros that allow the service developer to print debug statements in a configurable way. The debug statements can have different levels of severity, and are controlled by environment variables. Debug statements are only printed when the service module is compiled with a debug flavor (such as `gcc32dbg`).

The macro declaration for printing a debug statement in the service skeleton is:

```
<service>DebugPrintf(LEVEL, MESSAGE);
```

Where `LEVEL` is one of:

- `<SERVICE>_INFO`
- `<SERVICE>_DEBUG`

- `<SERVICE>_TRACE`
- `<SERVICE>_WARN`
- `<SERVICE>_ERROR`

The `MESSAGE` parameter consists of the debug message to be printed. It must contain parentheses `()` around the actual message. Inside the parentheses can be a format string, and a variable number of arguments (like `printf`). For example, in the `BlogService`'s `addEntry` skeleton implementation (`Blog_addEntry_impl`), the developer may want to see the entry to be added for debugging purposes. The following statement would print the debug message if the `DEBUG` level was turned on:

```
BlogServiceDebugPrintf(BLOGSERVICE_DEBUG,
                       ("ADD ENTRY:\n\tCOMMENT: %s\n\tAUTHOR: %s\n",
                        addEntry->Comment,
                        addEntry->Author));
```

## 2.7.2. Setting Debug Environment Variables

In order for debug statements to be printed to the terminal, the user must set the appropriate environment variable before running the service container. Each service has a separate debug environment variable that can be set to different debug levels. Optionally, the value of the variable can include a filename to write the debug output to as well.

The environment variable to set for service debugging is:

```
<SERVICE>_DEBUG=<DEBUG LEVEL>
```

This environment variable has five disjoint debug levels that can be set, and match the level definitions used for the debug statement in the previous section. The five levels are:

- `INFO` - general information useful to users of the service.
- `DEBUG` - debug output used by the service skeleton implementor to verify code works.
- `TRACE` - output the entry and exit points of each of the service skeleton functions.
- `WARN` - warn the user that something bad may be happening.
- `ERROR` - output an error for the user to see as it gets returned.

There is also a `ALL` level that will show the debug output for all the levels.

For our `BlogService` example, if we wanted to see the debug statements at the `DEBUG` level, then in `bash` we would set:

```
export BLOGSERVICE_DEBUG=DEBUG
```

If the user wants to see output from multiple debug levels, the levels can be joined together:

```
export BLOGSERVICE_DEBUG="DEBUG|TRACE"
```

---

# Chapter 4. Architecture and design overview

- [Mapping WSDL to C Bindings](#) - describes how to use the C bindings generated from WSDL and XML schema.
- [Design of Web Services Architecture in C](#)<sup>1</sup>
- [Design of WSRF in C](#)<sup>2</sup>

---

<sup>1</sup> ../C-GT4-WS-Design.pdf

<sup>2</sup> ../C-GT4-WSRF-Design.pdf

---

# Chapter 5. APIs

## 1. Programming Model Overview

The C WS-Core provides interfaces for developers interested in writing web services and clients in C. The primary APIs available to the developer are the C stub bindings generated from WSDL and XSD. These APIs provide the structures and type definitions for each XML Schema type, client stub functions for invoking services, and service skeleton code that allows service writers to fill in the service implementation.

The client stub bindings provide the following:

- Portable ANSI-C API
- Control of message handling and configurable attributes through client handles
- Asynchronous stub functions for non-blocking requests
- EPR encapsulation for easy interaction with resources
- Convenient handling of XSD wildcards

For service writers, the C WS-Core provides service-side skeleton bindings that perform the necessary routing and marshalling for a service operation. The interface to the developer is through the service implementation functions that must be filled in. The service-side programming model includes the ability to load operation providers, which are generic operation implementations that exist over a set of services. This is useful with WSRF, where pseudo-operation inheritance exists. As well, message handling can be controlled at the service implementation level, providing flexibility and control to the service developer.

The C WS-Core provides resource management using the resource API. This is a C API that can be invoked from within C services for creation, access, and control of resources and resource properties.

## 2. Component API

- Resource and Resource Property API<sup>1</sup>: Useful for writing WSRF-enabled services. This API allows resources to be created, accessed, and modified from within a C Web Service implementation.
- Service Engine and Message Attributes<sup>2</sup>: The message attributes provides mechanisms for manipulating runtime parameters of messages. This includes security setup, specific HTTP and WS-Addressing configuration, among others.

The service engine API is useful for embedding Web Services in C programs. This API allows an application to directly control service invocations and interact with services as they are being invoked. It also provides a convenient API for running a NotificationConsumer service (receiving notifications) from within a client application.

- Notification Consumer API<sup>3</sup>: Allows creation of NotificationConsumer resource instances from a client API. This API can be used in combination with the Service Engine API to receive notifications.

---

<sup>1</sup> [http://www.globus.org/api/c-globus-4.2.1/globus\\_c\\_wsrf\\_resource/html/index.html](http://www.globus.org/api/c-globus-4.2.1/globus_c_wsrf_resource/html/index.html)

<sup>2</sup> [http://www.globus.org/api/c-globus-4.2.1/globus\\_c\\_ws\\_messaging/html/index.html](http://www.globus.org/api/c-globus-4.2.1/globus_c_ws_messaging/html/index.html)

<sup>3</sup> [http://www.globus.org/api/c-globus-4.2.1/globus\\_notification\\_consumer/html/index.html](http://www.globus.org/api/c-globus-4.2.1/globus_notification_consumer/html/index.html)

- WSRF Core Bindings API<sup>4</sup>: These are the types generated from the set of core WSRF schemas. For example, the *wsa\_EndpointReferenceType* passed to all EPR stub functions is a generated type from the WS-Addressing schema. The other schemas include:
  - WS-Addressing
  - WS-BaseFaults
  - WS-ResourceProperties
  - WS-ResourceLifetime
  - WS-BaseN
  - WS-ServiceGroup

---

<sup>4</sup> [http://www.globus.org/api/c-globus-4.2.1/globus\\_c\\_wsrp\\_core\\_bindings/html/index.html](http://www.globus.org/api/c-globus-4.2.1/globus_c_wsrp_core_bindings/html/index.html)

---

# **C WS Core Commands**

---

# Name

globus-wsc-container -- Host C web services

```
globus-wsc-container [-help] [-usage] [-version]
[-bg] [-pidfile PID]
[-max MAX-SESSIONS]
[-port PORT]
[-log LOGPATH]
[-nosec]
```

# Description

The **globus-wsc-container** is a stand-alone SOAP service hosting container. It listens for SOAP / HTTP operation requests on a network port and dispatches those to dynamically loaded service modules. By default, **globus-wsc-container** will process SOAP messages until it receives a SIGINT or SIGTERM signal. In interactive usage, it typically runs until the user enters **Ctrl+C** on the keyboard.

The full set of command-line options to **globus-wsc-container** are:

-help	Display a help message and exit
-usage	Display a short usage message and exit
-version	Display the program version and exit
-bg	Run the program as a daemon
-pid <i>PIDFILE</i>	Write the process ID of the program to <i>PIDFILE</i>
-max <i>MAX-SESSIONS</i>	Allow at most <i>MAX-SESSIONS</i> concurrent sessions to be processed by the program
-port <i>PORT</i>	Listen for SOAP/HTTP(s) connections on TCP port <i>PORT</i>
-log <i>LOGPATH</i>	Log container information to <i>LOGPATH</i>
-nosec	Disable TLS

By default, the **globus-wsc-container** program picks an anonymous TCP port within values specified by the `GLOBUS_TCP_PORT_RANGE` environment variable, if present. To choose a specific port to listen on, pass the option `-port PORT` on the command-line of the process.

The **globus-wsc-container** program can also be run in the background as a daemon. This is done by passing the `-bg` command-line option. This can be combined with the `-pidfile PID` option to run in the background and record the PID of the process in a file, so that the daemon can be easily terminated.

By default, the container uses TLS for SOAP requests over https. This can be disabled to use unprotected http by passing the `-nosec` command-line option to this program. Message-level security may be enabled on a per-service basis if this is used.

To enable CEDPs "best practices" logging, pass the `-log LOGPATH` option to the container. The log file will contain name=value pairs for all events that the container processes.

By default the container will accept as many SOAP connections as the operating system will allow. To throttle the number of outstanding connections that can be processed in parallel, use the `-max MAX-SESSIONS` command-line option.

## Services

The container looks for services in dynamic modules located in the `$GLOBUS_LOCATION/lib/globus_service_modules` directory. The Globus Toolkit ships with a number of sample services, test services, and implementations of the core *WSRF* services for implementing Resource Properties, Resource Lifetime, Service Groups, and Notifications. The **globus-wsrf-cgen** command parses *WSDL* schemas and generates service skeletons which can be used to implement additional web services.

## Examples

Start a container in the foreground on port 8443:

```
% globus-wsc-container -port 8443
```

Contact: <https://grid.example.org:8443/>

Star a container as a daemon on an anonymous port, with a maximum of 64 parallel sessions, recording the port number to a file and logging to another file.

```
% globus-wsc-container \  
  -bg \  
  -pidfile $GLOBUS_LOCATION/var/globus-wsc-container.pid \  
  -log $GLOBUS_LOCATION/var/globus-wsc-container.log \  
  -max 64  
  > $GLOBUS_LOCATION/var/globus-wsc-container.contact
```

```
% cat $GLOBUS_LOCATION/var/globus-wsc-container.contact
```

Contact: <https://grid.example.org:18332/>

```
% cat $GLOBUS_LOCATION/var/globus-wsc-container.log
```

```
ts=2008-06-19T22:43:21.645807Z id=21475 event=globus_service_engine.start engine_id=40235
```

---

# Name

globus-wsrf-cgen -- Generate Stubs/Skeletons in C

```
globus-wsrf-cgen [-help] [-dr]
[-s PACKAGE-NAME] [-sn SERVICE-NAME] [-d DIRECTORY] [-flavor FLAVOR] [-lang [ c | cpp ]]
[-p PREFIX-MAP-FILE] [-P NAMESPACE=PREFIX]
[-n NAMESPACE-FILE] [-N NAMESPACE]
[-g NAMESPACE-FILE] [-G NAMESPACE] [-gg]
[-np] [-nb] [-nk] [-ns] [-nc] [-no-sources] [-nt] [-nf FUNCTION]
[-extra-cppflags CPPFLAGS] [-extra-ldflags LDFLAGS] [-extra-libs LIBS]
SCHEMA-FILENAME...
```

## Description

The **globus-wsrf-cgen** tool generates C-language bindings from WSDL and XML Schema files. The input *SCHEMA-FILENAME* value should be either a WSDL document containing a service description or an XML schema file containing type definitions.

If a WSDL Schema file is specified as input, **globus-wsrf-cgen** generates a GPT source package containing client stubs, service skeleton and stubs, and type bindings for included schema types. If an XML Schema file is specified as input, it generates a GPT source package containing type bindings. A full description of the generated files is part of the [WSDL to C mapping document](#).

The full set of command-line options to **globus-wsrf-cgen** are:

-help	Display a help message and exit
-dr	Dry-run: parse the command-line options and display the command-line arguments to the <b>globus-wsdl-parser</b> program.
-s <i>PACKAGE-NAME</i>	Use <i>PACKAGE-NAME</i> _bindings as the name for the generated GPT package
-sn <i>SERVICE-NAME</i>	Use <i>SERVICE-NAME</i> as the name of the service instead of the name in the WSDL schema document.
-d <i>DIRECTORY</i>	Generate the GPT source package in <i>DIRECTORY</i> , creating it if does not exist.
-flavor <i>FLAVOR</i>	Build the package using the <i>FLAVOR</i> GPT <i>flavor</i>
-lang <i>LANG</i>	Create the service implementation file with the extension matching <i>LANG</i> , either "c" or "cpp". See the <a href="#">limitations</a> section for more details.
-p <i>PREFIX-MAP-FILE</i>	Use the contents of <i>PREFIX-MAP-FILE</i> to define the set of strings to prepend to elements, attributes, and types in various XML namespaces. See the <a href="#">namespace handling</a> section of this document for more details.
-P <i>NAMESPACE=PREFIX-FILE</i>	Prepend element, attribute, and type names in the XML namespace <i>NAMESPACE</i> with the string <i>PREFIX</i> . See the <a href="#">namespace handling</a> section of this document for more details.
-n <i>NAMESPACE-FILE</i>	Generate bindings for schemas in the XML namespaces contained in the <i>NAMESPACE-FILE</i> . See the <a href="#">namespace handling</a> section of this document for more details.
-N <i>NAMESPACE</i>	Generate bindings for schemas in the XML namespace <i>NAMESPACE</i> . See the <a href="#">namespace handling</a> section of this document for more details.

<code>-g <i>NAMESPACE-FILE</i></code>	Do not generate bindings for schemas in the XML namespaces contained in the <i>NAMESPACE-FILE</i> . See the <a href="#">namespace handling</a> section of this document for more details.
<code>-G <i>NAMESPACE</i></code>	Do not generate bindings for schemas in the XML namespace <i>NAMESPACE</i> . See the <a href="#">namespace handling</a> section of this document for more details.
<code>-gg</code>	Do not generate bindings for core WSRF namespaces. (Used internally only)
<code>-np</code>	Do not generate a GPT package. Only create source files from the schemas. Implies <code>-nb</code> .
<code>-nb</code>	Do not attempt to run <b>configure</b> and <b>make dist</b> on the generated GPT source package.
<code>-nk</code>	Do not generate a skeleton service implementation. Used in Makefiles for packages that want to generate the types at build time, but already contain a full implementation of the service.
<code>-ns</code>	Do not generate service bindings and skeletons. Useful for creating types- or client-only packages.
<code>-nc</code>	Do not generate client bindings. Useful for creating types- or service-only packages.
<code>-nt</code>	Do not generate type bindings. Useful for creating separate service or client bindings that depend on a common types package.
<code>-no-sources</code>	Delay generating C source files until the package is built. By default the package Makefile contains a list of source files. This option delays the creation of the files and the list until build time. This can be used to avoid storing dynamic files in a version control system.
<code>-nf <i>FUNCTION</i></code>	Do not generate an implementation of <i>FUNCTION</i> . This is useful if extra semantic information is needed to serialize or deserialize a particular data type (for example, the <code>wsnt:TopicExpressionType</code> requires different processing based on the value of the <code>Dialect</code>
<code>-extra-cppflags <i>CPP-FLAGS</i></code>	Add <i>CPPFLAGS</i> to the preprocessor command-line for this package.
<code>-extra-ldflags <i>LDFLAGS</i></code>	Add <i>LDFLAGS</i> to the linker command-line for this package.
<code>-extra-libs <i>LIBS</i></code>	Add <i>LIBS</i> to the libraries to link with this package.

## Namespace Handling

XML and WSDL schemas generally contain a `targetNamespace` attribute which distinguishes operations, elements, attributes, type, etc from others with the same name. The C language does not define namespaces. **globus-wsrf-cgen** instead uses prefixes to distinguish similarly-named data types and functions. There are two ways to define a namespace prefix with **globus-wsrf-cgen**. The `-P` command-line option defines a single namespace prefix, and the `-p` command-line option instructs **globus-wsrf-cgen** to load a set of prefix definitions from a file (one per line).

For example, consider the namespace `http://counter.com` from the sample `CounterService`. In the schema for that service, there is an element named `Value`. the command-line option `-P http://counter.com=counter_` will cause **globus-wsrf-cgen** to generate bindings for that element with the name `counter_Value`.

If a service is built from several namespaces it might make sense instead to use the `-P` parameter instead. Using the same service as the previous example, we could instead create a file containing

```
http://counter.com=counter_
http://another.counter.com=another_counter_
```

to generate C prefixes for multiple namespaces.

A service may be composed of operations and data types from multiple namespaces. By default **globus-wsrf-cgen** generates bindings for all namespaces except those used by the core WSRF specifications. These are (along with their C prefixes):

**Table 1. WSRF Core Namespaces and C Prefixes**

http://www.w3.org/XML/1998/namespace	xml_
http://www.w3.org/2001/XMLSchema	xsd_
http://www.w3.org/2005/08/addressing	wsa_
http://docs.oasis-open.org/wsrf/r-2	wsr_
http://docs.oasis-open.org/wsrf/rw-2	wsrw_
http://docs.oasis-open.org/wsrf/bf-2	wsbf_
http://docs.oasis-open.org/wsrf/rp-2	wsrp_
http://docs.oasis-open.org/wsrf/rpw-2	wsrpw_
http://docs.oasis-open.org/wsrf/rl-2	wsrl_
http://docs.oasis-open.org/wsrf/rlw-2	wsrlw_
http://docs.oasis-open.org/wsrf/sg-2	wssg_
http://docs.oasis-open.org/wsrf/sgw-2	wssgw_
http://docs.oasis-open.org/wsn/b-2	wsnt_
http://docs.oasis-open.org/wsn/bw-2	wsntw_
http://docs.oasis-open.org/wsn/t-1	wstop_
http://schemas.xmlsoap.org/ws/2002/12/policy	wsp_
http://schemas.xmlsoap.org/ws/2002/07/utility	wsu_
http://schemas.xmlsoap.org/ws/2004/04/trust	wst_
http://www.w3.org/2000/09/xmlsig#	ds_
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd	wsse_
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd	wsseu_
http://schemas.xmlsoap.org/ws/2004/04/sc	wsc_
http://schemas.xmlsoap.org/ws/2004/09/enumeration	wsen_

Often it is enough for a package to contain bindings for the namespaces unique to the service and compile against other packages which contain the bindings for the other namespaces. This control can be done positively via the `-N` and `-n` command-line options.

For example, to generate bindings for the `http://counter.com` namespace *only*, pass the command-line option `-N http://counter.com`. To generate for both the `http://counter.com` and `http://another.counter.com` namespaces, either pass multiple `-N` options with one namespace each, or create a file containing:

```
http://counter.com
http://another.counter.com
```

and pass the name of the file to `globus-wsrf-cgen` as the parameter to the `-n` command-line option.

## Examples

Here is a brief example of the **globus-wsrf-cgen** command. For more details, see the [tutorials in the C WS Core developer documentation](#).

Create bindings for a service in the http://counter.com namespace:

```
% globus-wsrf-cgen -d counter \  
  -N http://counter.com \  
  -s counter \  
  -P http://counter.com=counter_ \  
  $GLOBUS_LOCATION/share/schemas/core/samples/counter_service.wsdl
```

### Creating Bindings Package

A new package has been created at /home/griduser/counter/counter\_bindings-1.2.tar.gz  
To install, use the following command:

```
$GLOBUS_LOCATION/sbin/gpt-build /Users/bester/tmp/foo/counter/counter_bindings-1.2.tar.gz  
%
```

## Limitations

- This program only generates bindings from document/literal style WSDL schemas. IBM developerworks has [an article describing the different WSDL schema styles](#)<sup>1</sup>.
- The bindings generated when `-lang cpp` is used are ANSI-C. However, all C++ keywords are avoided and no constructs that differ between C and C++ are used. This command-line option merely creates a makefile which compiles the service implementation with the C++ compiler.
- Not all XML Schema constructs are supported. In particular, abstract types, substitution groups, and nested sequences are not implemented.

---

<sup>1</sup> <http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/>

---

# Name

globus-wsrf-destroy -- Set the scheduled termination time for a WSRF resource.

globus-wsrf-destroy [OPTIONS]... SERVICE-SPECIFIER

## Tool description

Set the scheduled termination time for a WSRF resource.

## Command syntax

globus-wsrf-destroy [OPTIONS]... SERVICE-SPECIFIER

**Table 2. Common options**

<b>-a   --anonymous</b>	Use anonymous authentication. Requires either <b>-m 'conv'</b> or transport (https) security.
<b>-d, --debug</b>	Enables debug mode. In debug mode, all SOAP messages will be displayed to stderr and full WSRF Fault messages will be displayed.
<b>-e   --eprFile FILENAME</b>	Load service EPR from FILENAME. This EPR is used to contact the WSRF service.
<b>-h   --help</b>	Displays help information about the command.
<b>-k   --key KEYNAME VALUE</b>	Set resource key in the service EPR to be named KEYNAME with VALUE as its value. This can be combined with <b>-s</b> to construct an EPR without having an xml file on hand. The <b>KEYNAME</b> is a QName string in the format <b>{namespaceURI}localPart</b> , while the <b>VALUE</b> is a literal string to place in the element. For example, the option <b>-k '{http://www.globus.org}MyKey' 128</b> would be rendered as <b>&lt;MyKey xmlns="http://www.globus.org"&gt;128&lt;/MyKey&gt;</b>
<b>-m, --securityMech TYPE</b>	Set authentication mechanism. TYPE is one of <b>msg</b> for WS-SecureMessage or <b>conv</b> for WS-SecureConversation.
<b>-p, --protection LEVEL</b>	Set message protection level. LEVEL is one of <b>sig</b> for digital signature or <b>enc</b> for encryption. The default is 'sig'.
<b>-s   --service ENDPOINT</b>	Set ENDPOINT the service URL to use. Will be composed with the <b>-k</b> parameter if present to add ReferenceProperties to the ENDPOINT
<b>-t   --timeout SECONDS</b>	Set client timeout to SECONDS.
<b>-u   --usage</b>	Print short usage message.
<b>-V   --version</b>	Show version information and exit.
<b>-v   --certKeyFiles CERTIFICATE-FILENAME KEY-FILENAME</b>	Use credentials located in CERTIFICATE-FILENAME and KEY-FILENAME. The key file must be unencrypted.
<b>-x   --proxyFilename FILENAME</b>	Use proxy credentials located in FILENAME.
<b>-z   --authorization TYPE</b>	Set authorization mode. TYPE can be <b>self</b> , <b>host</b> , <b>none</b> , or a string specifying the identity of the remote party. The default is <b>self</b> .
<b>--versions</b>	Show version information for all loaded modules and exit.

SERVICE-SPECIFIER: [-s URI [-k KEY VALUE] | -e FILENAME]

*Examples:*

```
% globus-wsrf-destroy -e widget.epr  
Resource destroyed
```

Contents of *widget.epr*:

```
<ns01:EndpointReference xmlns:ns01="http://schemas.xmlsoap.org/ws/2004/03/addressing">  
  <ns01:Address>http://globus.my.org:8080/wsrf/services/WidgetService</ns01:Address>  
  <ns01:ReferenceProperties>  
    <ResourceID xmlns:ns02="http://www.w3.org/2001/XMLSchema-instance" xmlns:ns03="http://"  
  </ns01:ReferenceProperties>  
</ns01:EndpointReference>
```

## Output and Exit Code

If the resource is destroyed successfully, the string `Resource destroyed` will be displayed to *stdout* and the program will terminate with exit code 0. In the case of an error, the type of error will be displayed to *stderr* and the program will terminate with a non-0 exit code.

---

# Name

globus-wsrf-set-termination-time -- Set the scheduled termination time for a WSRF resource.

```
globus-wsrf-set-termination-time [OPTIONS]... SERVICE-SPECIFIER TERMINATION-TIME
```

## Tool description

Set the scheduled termination time for a WSRF resource.

## Command syntax

```
globus-wsrf-set-termination-time [OPTIONS]... SERVICE-SPECIFIER TERMINATION-TIME
```

**Table 3. Common options**

<b>-a   --anonymous</b>	Use anonymous authentication. Requires either <b>-m 'conv'</b> or transport (https) security.
<b>-d, --debug</b>	Enables debug mode. In debug mode, all SOAP messages will be displayed to stderr and full WSRF Fault messages will be displayed.
<b>-e   --eprFile FILENAME</b>	Load service EPR from FILENAME. This EPR is used to contact the WSRF service.
<b>-h   --help</b>	Displays help information about the command.
<b>-k   --key KEYNAME VALUE</b>	Set resource key in the service EPR to be named KEYNAME with VALUE as its value. This can be combined with <b>-s</b> to construct an EPR without having an xml file on hand. The <b>KEYNAME</b> is a QName string in the format <b>{namespaceURI}localPart</b> . while the <b>VALUE</b> is a literal string to place in the element. For example, the option <b>-k '{http://www.globus.org}MyKey' 128</b> would be rendered as <b>&lt;MyKey xmlns="http://www.globus.org"&gt;128&lt;/MyKey&gt;</b>
<b>-m, --securityMech TYPE</b>	Set authentication mechanism. TYPE is one of <b>msg</b> for WS-SecureMessage or <b>conv</b> for WS-SecureConversation.
<b>-p, --protection LEVEL</b>	Set message protection level. LEVEL is one of <b>sig</b> for digital signature or <b>enc</b> for encryption. The default is 'sig'.
<b>-s   --service ENDPOINT</b>	Set ENDPOINT the service URL to use. Will be composed with the <b>-k</b> parameter if present to add ReferenceProperties to the ENDPOINT
<b>-t   --timeout SECONDS</b>	Set client timeout to SECONDS.
<b>-u   --usage</b>	Print short usage message.
<b>-V   --version</b>	Show version information and exit.
<b>-v   --certKeyFiles CERTIFICATE-FILENAME KEY-FILENAME</b>	Use credentials located in <b>CERTIFICATE-FILENAME</b> and <b>KEY-FILENAME</b> . The key file must be unencrypted.
<b>-x   --proxyFilename FILENAME</b>	Use proxy credentials located in <b>FILENAME</b> .
<b>-z   --authorization TYPE</b>	Set authorization mode. <b>TYPE</b> can be <b>self</b> , <b>host</b> , <b>none</b> , or a string specifying the identity of the remote party. The default is <b>self</b> .
<b>--versions</b>	Show version information for all loaded modules and exit.

SERVICE-SPECIFIER: [-s URI [-k KEY VALUE] | -e FILENAME]

TERMINATION-TERMINATION: [SECONDS | 'infinity']

*Examples:*

```
% globus-wsrf-set-termination-time -e widget.epr `expr 24 \* 60 \* 60`  
Termination time set to 2006-05-31T20:18:43Z
```

Contents of *widget.epr*:

```
<ns01:EndpointReference xmlns:ns01="http://schemas.xmlsoap.org/ws/2004/03/addressing">  
  <ns01:Address>http://globus.my.org:8080/wsrf/services/WidgetService</ns01:Address>  
  <ns01:ReferenceProperties>  
    <ResourceID xmlns:ns02="http://www.w3.org/2001/XMLSchema-instance" xmlns:ns03="http://"  
  </ns01:ReferenceProperties>  
</ns01:EndpointReference>
```

## Output and Exit Code

If the termination time is set successfully, the string Termination time set to YYYY-MM-DD-THH:MM:SS[.MSEC]Z will be displayed to *stdout* and the program will terminate with exit code 0. In the case of an error, the type of error will be displayed to *stderr* and the program will terminate with a non-0 exit code.

---

## Name

globus-wsrf-query -- Query a WSRF resource's Resource Property document

globus-wsrf-query [OPTIONS]... SERVICE-SPECIFIER QUERY-EXPRESSION

## Tool description

Perform an XPATH query on a resource property document.

## Command syntax

globus-wsrf-query [OPTIONS]... SERVICE-SPECIFIER QUERY-EXPRESSION

**Table 4. Application-specific options**

<b>-n   ---nsMapFile FILENAME.</b>	Use the namespace map entries in <i>FILENAME</i> in the XPATH context.
<b>-N   --namespace PREFIX=NAMESPACE-URI</b>	Create a namespace mapping in the XPATH context for the <i>PREFIX</i> string to resolve to the <i>NAMESPACE-URI</i> namespace.
<b>-D   --dialect DIALECT-URI</b>	Set query dialect to <i>DIALECT-URI</i> . The value <b>targeted</b> will be interpreted as <a href="http://wsrf.globus.org/core/query/targetedXPath">http://wsrf.globus.org/core/query/targetedXPath</a> (default: <a href="http://www.w3.org/TR/1999/REC-xpath-19991116">http://www.w3.org/TR/1999/REC-xpath-19991116</a> ).

**Table 5. Common options**

<b>-a   --anonymous</b>	Use anonymous authentication. Requires either <b>-m 'conv'</b> or transport (https) security.
<b>-d, --debug</b>	Enables debug mode. In debug mode, all SOAP messages will be displayed to stderr and full WSRF Fault messages will be displayed.
<b>-e   --eprFile FILENAME</b>	Load service EPR from FILENAME. This EPR is used to contact the WSRF service.
<b>-h   --help</b>	Displays help information about the command.
<b>-k   --key KEYNAME VALUE</b>	Set resource key in the service EPR to be named KEYNAME with VALUE as its value. This can be combined with <b>-s</b> to construct an EPR without having an xml file on hand. The <b>KEYNAME</b> is a QName string in the format <b>{namespaceURI}localPart</b> . while the <b>VALUE</b> is a literal string to place in the element. For example, the option <b>-k '{http://www.globus.org}MyKey' 128</b> would be rendered as <b>&lt;MyKey xmlns="http://www.globus.org"&gt;128&lt;/MyKey&gt;</b>
<b>-m, --securityMech TYPE</b>	Set authentication mechanism. TYPE is one of <b>msg</b> for WS-SecureMessage or <b>conv</b> for WS-SecureConversation.
<b>-p, --protection LEVEL</b>	Set message protection level. LEVEL is one of <b>sig</b> for digital signature or <b>enc</b> for encryption. The default is 'sig'.
<b>-s   --service ENDPOINT</b>	Set ENDPOINT the service URL to use. Will be composed with the <b>-k</b> parameter if present to add ReferenceProperties to the ENDPOINT
<b>-t   --timeout SECONDS</b>	Set client timeout to SECONDS.
<b>-u   --usage</b>	Print short usage message.
<b>-V   --version</b>	Show version information and exit.
<b>-v   --certKeyFiles CERTIFICATE-FILENAME KEY-FILENAME</b>	Use credentials located in <b>CERTIFICATE-FILENAME</b> and <b>KEY-FILENAME</b> . The key file must be unencrypted.
<b>-x   --proxyFilename FILENAME</b>	Use proxy credentials located in <b>FILENAME</b> .
<b>-z   --authorization TYPE</b>	Set authorization mode. <b>TYPE</b> can be <b>self</b> , <b>host</b> , <b>none</b> , or a string specifying the identity of the remote party. The default is <b>self</b> .
<b>--versions</b>	Show version information for all loaded modules and exit.

SERVICE-SPECIFIER: [-s URI [-k KEY VALUE] | -e FILENAME]

QUERY-EXPRESSION: XPath-Expression-String

*Examples:*

```
% globus-wsrf-query -e widget.epr "//*[local-name() = 'CurrentTime']"
<ns0:CurrentTime
  xmlns:ns0="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ns1="http://www.w3.org/2001/XMLSchema"
  xmlns:ns2="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft
  ns0:type="ns1:dateTime">2006-05-30T13:53:15Z</ns0:CurrentTime>
```

```
% globus-wsrf-query -e widget.epr "//*[local-name() = 'CurrentTime']/text()"
2006-05-30T13:53:35Z
```

```
% globus-wsrf-query -e widget.epr \
    -N wsrl=http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-
    "/wsrl:CurrentTime/text()"
2006-05-30T13:54:36Z
```

Contents of *widget.epr*:

```
<ns01:EndpointReference xmlns:ns01="http://schemas.xmlsoap.org/ws/2004/03/addressing">
  <ns01:Address>http://globus.my.org:8080/wsrf/services/WidgetService</ns01:Address>
  <ns01:ReferenceProperties>
    <ResourceID xmlns:ns02="http://www.w3.org/2001/XMLSchema-instance" xmlns:ns03="http://
  </ns01:ReferenceProperties>
</ns01:EndpointReference>
```

## Limitations

- The namespace mapping option and use of namespace prefixes in the *XPath-Expression-String* does not work when communicating with the Java container unless the *http://wsrf.globus.org/core/query/targetedXPath* dialect is used.

## Output and Exit Code

If the query is successful, the program displays the output of the query to *stdout* and terminates with exit code 0. In the case of an error, the type of error will be displayed to *stderr* and the program will terminate with a non-0 exit code.

---

# Name

globus-wsrf-get-property -- Get a resource property's value

globus-wsrf-get-property [OPTIONS]... SERVICE-SPECIFIER PROPERTY-NAME

## Tool description

Get the value of a resource property from a WSRF resource.

## Command syntax

globus-wsrf-get-property [OPTIONS]... SERVICE-SPECIFIER PROPERTY-NAME

**Table 6. Common options**

<b>-a   --anonymous</b>	Use anonymous authentication. Requires either <b>-m 'conv'</b> or transport (https) security.
<b>-d, --debug</b>	Enables debug mode. In debug mode, all SOAP messages will be displayed to stderr and full WSRF Fault messages will be displayed.
<b>-e   --eprFile FILENAME</b>	Load service EPR from FILENAME. This EPR is used to contact the WSRF service.
<b>-h   --help</b>	Displays help information about the command.
<b>-k   --key KEYNAME VALUE</b>	Set resource key in the service EPR to be named KEYNAME with VALUE as its value. This can be combined with <b>-s</b> to construct an EPR without having an xml file on hand. The <b>KEYNAME</b> is a QName string in the format <b>{namespaceURI}localPart</b> , while the <b>VALUE</b> is a literal string to place in the element. For example, the option <b>-k '{http://www.globus.org}MyKey' 128</b> would be rendered as <b>&lt;MyKey xmlns="http://www.globus.org"&gt;128&lt;/MyKey&gt;</b>
<b>-m, --securityMech TYPE</b>	Set authentication mechanism. TYPE is one of <b>msg</b> for WS-SecureMessage or <b>conv</b> for WS-SecureConversation.
<b>-p, --protection LEVEL</b>	Set message protection level. LEVEL is one of <b>sig</b> for digital signature or <b>enc</b> for encryption. The default is 'sig'.
<b>-s   --service ENDPOINT</b>	Set ENDPOINT the service URL to use. Will be composed with the <b>-k</b> parameter if present to add ReferenceProperties to the ENDPOINT
<b>-t   --timeout SECONDS</b>	Set client timeout to SECONDS.
<b>-u   --usage</b>	Print short usage message.
<b>-V   --version</b>	Show version information and exit.
<b>-v   --certKeyFiles CERTIFICATE-FILENAME KEY-FILENAME</b>	Use credentials located in CERTIFICATE-FILENAME and KEY-FILENAME. The key file must be unencrypted.
<b>-x   --proxyFilename FILENAME</b>	Use proxy credentials located in FILENAME.
<b>-z   --authorization TYPE</b>	Set authorization mode. TYPE can be <b>self</b> , <b>host</b> , <b>none</b> , or a string specifying the identity of the remote party. The default is <b>self</b> .
<b>--versions</b>	Show version information for all loaded modules and exit.

SERVICE-SPECIFIER: [-s URI [-k KEY VALUE] | -e FILENAME]

PROPERTY-NAME: [{Namespace-URI}]Property-Name

*Example:*

```
% globus-wsrf-get-property -e widget.epr \  
    '{http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-01.xsd  
  
<ns02:CurrentTime  
    xmlns:ns00="http://www.w3.org/2001/XMLSchema-instance"  
    xmlns:ns01="http://www.w3.org/2001/XMLSchema"  
    xmlns:ns02="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft  
    ns00:type="ns01:dateTime">2006-05-30T14:26:35Z</ns02:CurrentTime>
```

## Output and Exit Code

If the property exists, its values (if any) are displayed to *stdout* and the program terminates with exit code 0. In the case of an error, the type of error will be displayed to *stderr* and the program will terminate with a non-0 exit code.

---

# Name

globus-wsrf-get-properties -- Get multiple resource property value

globus-wsrf-get-properties [OPTIONS]... SERVICE-SPECIFIER PROPERTY-NAME...

## Tool description

Get the value of multiple resource properties from a WSRF resource.

## Command syntax

globus-wsrf-get-properties [OPTIONS]... SERVICE-SPECIFIER PROPERTY-NAME...

**Table 7. Common options**

<b>-a   --anonymous</b>	Use anonymous authentication. Requires either <b>-m 'conv'</b> or transport (https) security.
<b>-d, --debug</b>	Enables debug mode. In debug mode, all SOAP messages will be displayed to stderr and full WSRF Fault messages will be displayed.
<b>-e   --eprFile FILENAME</b>	Load service EPR from FILENAME. This EPR is used to contact the WSRF service.
<b>-h   --help</b>	Displays help information about the command.
<b>-k   --key KEYNAME VALUE</b>	Set resource key in the service EPR to be named KEYNAME with VALUE as its value. This can be combined with <b>-s</b> to construct an EPR without having an xml file on hand. The <b>KEYNAME</b> is a QName string in the format <b>{namespaceURI}localPart</b> . while the <b>VALUE</b> is a literal string to place in the element. For example, the option <b>-k '{http://www.globus.org}MyKey' 128</b> would be rendered as <b>&lt;MyKey xmlns="http://www.globus.org"&gt;128&lt;/MyKey&gt;</b>
<b>-m, --securityMech TYPE</b>	Set authentication mechanism. TYPE is one of <b>msg</b> for WS-SecureMessage or <b>conv</b> for WS-SecureConversation.
<b>-p, --protection LEVEL</b>	Set message protection level. LEVEL is one of <b>sig</b> for digital signature or <b>enc</b> for encryption. The default is 'sig'.
<b>-s   --service ENDPOINT</b>	Set ENDPOINT the service URL to use. Will be composed with the <b>-k</b> parameter if present to add ReferenceProperties to the ENDPOINT
<b>-t   --timeout SECONDS</b>	Set client timeout to SECONDS.
<b>-u   --usage</b>	Print short usage message.
<b>-V   --version</b>	Show version information and exit.
<b>-v   --certKeyFiles CERTIFICATE-FILENAME KEY-FILENAME</b>	Use credentials located in CERTIFICATE-FILENAME and KEY-FILENAME. The key file must be unencrypted.
<b>-x   --proxyFilename FILENAME</b>	Use proxy credentials located in FILENAME.
<b>-z   --authorization TYPE</b>	Set authorization mode. TYPE can be <b>self</b> , <b>host</b> , <b>none</b> , or a string specifying the identity of the remote party. The default is <b>self</b> .
<b>--versions</b>	Show version information for all loaded modules and exit.

SERVICE-SPECIFIER: [-s URI [-k KEY VALUE] | -e FILENAME]

PROPERTY-NAME: [{Namespace-URI}]Property-Name

*Example:*

```
% globus-wsrf-get-properties \
  -s http://grid.example.org:8080/wsrf/services/WidgetService \
  -k "{http://www.globus.org/namespaces/2004/06/core}WidgetKey" 123 \
  "{http://widgets.com}foo" \
  "{http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-01.xsd}foo"
<ns02:foo
  xmlns:ns0="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ns1="http://www.w3.org/2001/XMLSchema"
  xmlns:ns2="http://widgets.com"
  ns0:type="ns01:string">
Foo Value String
</ns02:foo><ns03:CurrentTime
  xmlns:ns0="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ns1="http://www.w3.org/2001/XMLSchema"
  xmlns:ns3="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-01.xsd"
  ns0:type="ns01:dateTime">2006-05-30T16:04:15Z</ns03:CurrentTime>
```

## Output and Exit Code

If the properties exist, their values (if any) are displayed to *stdout* and the program terminates with exit code 0. In the case of an error, the type of error will be displayed to *stderr* and the program will terminate with a non-0 exit code.

---

## Name

globus-wsrf-insert-property -- Insert a resource property value

```
globus-wsrf-insert-property [OPTIONS]... SERVICE-SPECIFIER PROPERTY-VALUE-FILE-NAME
```

## Tool description

Insert a resource property into a WSRF resource's Resource Properties document. The new property will be read from the XML file specified by *PROPERTY-VALUE-FILENAME*.

## Command syntax

```
globus-wsrf-insert-property [OPTIONS]... SERVICE-SPECIFIER PROPERTY-VALUE-FILENAME...
```

**Table 8. Common options**

<b>-a   --anonymous</b>	Use anonymous authentication. Requires either <b>-m 'conv'</b> or transport (https) security.
<b>-d, --debug</b>	Enables debug mode. In debug mode, all SOAP messages will be displayed to stderr and full WSRF Fault messages will be displayed.
<b>-e   --eprFile FILENAME</b>	Load service EPR from FILENAME. This EPR is used to contact the WSRF service.
<b>-h   --help</b>	Displays help information about the command.
<b>-k   --key KEYNAME VALUE</b>	Set resource key in the service EPR to be named KEYNAME with VALUE as its value. This can be combined with <b>-s</b> to construct an EPR without having an xml file on hand. The <b>KEYNAME</b> is a QName string in the format <b>{namespaceURI}localPart</b> . while the <b>VALUE</b> is a literal string to place in the element. For example, the option <b>-k '{http://www.globus.org}MyKey' 128</b> would be rendered as <b>&lt;MyKey xmlns="http://www.globus.org"&gt;128&lt;/MyKey&gt;</b>
<b>-m, --securityMech TYPE</b>	Set authentication mechanism. TYPE is one of <b>msg</b> for WS-SecureMessage or <b>conv</b> for WS-SecureConversation.
<b>-p, --protection LEVEL</b>	Set message protection level. LEVEL is one of <b>sig</b> for digital signature or <b>enc</b> for encryption. The default is 'sig'.
<b>-s   --service ENDPOINT</b>	Set ENDPOINT the service URL to use. Will be composed with the <b>-k</b> parameter if present to add ReferenceProperties to the ENDPOINT
<b>-t   --timeout SECONDS</b>	Set client timeout to SECONDS.
<b>-u   --usage</b>	Print short usage message.
<b>-V   --version</b>	Show version information and exit.
<b>-v   --certKeyFiles CERTIFICATE-FILENAME KEY-FILENAME</b>	Use credentials located in <b>CERTIFICATE-FILENAME</b> and <b>KEY-FILENAME</b> . The key file must be unencrypted.
<b>-x   --proxyFilename FILENAME</b>	Use proxy credentials located in <b>FILENAME</b> .
<b>-z   --authorization TYPE</b>	Set authorization mode. <b>TYPE</b> can be <b>self</b> , <b>host</b> , <b>none</b> , or a string specifying the identity of the remote party. The default is <b>self</b> .
<b>--versions</b>	Show version information for all loaded modules and exit.

SERVICE-SPECIFIER: [-s URI [-k KEY VALUE] | -e FILENAME]

*Example:*

```
% globus-wsrf-insert-property -e widget.epr widget:foo.xml
```

Contents of *widget.epr*:

```
<ns01:EndpointReference xmlns:ns01="http://schemas.xmlsoap.org/ws/2004/03/addressing">
  <ns01:Address>http://globus.my.org:8080/wsrf/services/WidgetService</ns01:Address>
  <ns01:ReferenceProperties>
    <ResourceID xmlns:ns02="http://www.w3.org/2001/XMLSchema-instance" xmlns:ns03="http://
  </ns01:ReferenceProperties>
```

```
</ns01:EndpointReference>
```

Contents of *widget:foo.xml*:

```
<doc>
  <foo xmlns="http://widgets.com"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:type="xsd:string">
    Foo Value String
  </foo>
</doc>
```

## Output and Exit Code

If the property is inserted successfully, the program terminates with exit code 0. In the case of an error, the type of error will be displayed to *stderr* and the program will terminate with a non-0 exit code.

---

## Name

globus-wsrf-update-property -- Update a resource property value

```
globus-wsrf-update-property [OPTIONS]... SERVICE-SPECIFIER PROPERTY-VALUE-FILE-NAME
```

## Tool description

Update a resource property in a WSRF resource's Resource Properties document. The property's new value will be read from the XML file specified by *PROPERTY-VALUE-FILENAME*. An update operation will replace the value(s) of the resource property with the new value(s) in the property file.

## Command syntax

```
globus-wsrf-update-property [OPTIONS]... SERVICE-SPECIFIER PROPERTY-VALUE-FILENAME
```

**Table 9. Common options**

<b>-a   --anonymous</b>	Use anonymous authentication. Requires either <b>-m 'conv'</b> or transport (https) security.
<b>-d, --debug</b>	Enables debug mode. In debug mode, all SOAP messages will be displayed to stderr and full WSRF Fault messages will be displayed.
<b>-e   --eprFile FILENAME</b>	Load service EPR from FILENAME. This EPR is used to contact the WSRF service.
<b>-h   --help</b>	Displays help information about the command.
<b>-k   --key KEYNAME VALUE</b>	Set resource key in the service EPR to be named KEYNAME with VALUE as its value. This can be combined with <b>-s</b> to construct an EPR without having an xml file on hand. The <b>KEYNAME</b> is a QName string in the format <b>{namespaceURI}localPart</b> . while the <b>VALUE</b> is a literal string to place in the element. For example, the option <b>-k '{http://www.globus.org}MyKey' 128</b> would be rendered as <b>&lt;MyKey xmlns="http://www.globus.org"&gt;128&lt;/MyKey&gt;</b>
<b>-m, --securityMech TYPE</b>	Set authentication mechanism. TYPE is one of <b>msg</b> for WS-SecureMessage or <b>conv</b> for WS-SecureConversation.
<b>-p, --protection LEVEL</b>	Set message protection level. LEVEL is one of <b>sig</b> for digital signature or <b>enc</b> for encryption. The default is 'sig'.
<b>-s   --service ENDPOINT</b>	Set ENDPOINT the service URL to use. Will be composed with the <b>-k</b> parameter if present to add ReferenceProperties to the ENDPOINT
<b>-t   --timeout SECONDS</b>	Set client timeout to SECONDS.
<b>-u   --usage</b>	Print short usage message.
<b>-V   --version</b>	Show version information and exit.
<b>-v   --certKeyFiles CERTIFICATE-FILENAME KEY-FILENAME</b>	Use credentials located in <b>CERTIFICATE-FILENAME</b> and <b>KEY-FILENAME</b> . The key file must be unencrypted.
<b>-x   --proxyFilename FILENAME</b>	Use proxy credentials located in <b>FILENAME</b> .
<b>-z   --authorization TYPE</b>	Set authorization mode. <b>TYPE</b> can be <b>self</b> , <b>host</b> , <b>none</b> , or a string specifying the identity of the remote party. The default is <b>self</b> .
<b>--versions</b>	Show version information for all loaded modules and exit.

SERVICE-SPECIFIER: [-s URI [-k KEY VALUE] | -e FILENAME]

*Example:*

```
% globus-wsrf-update-property -e widget.epr widget:foo.xml
```

Contents of *widget.epr*:

```
<ns01:EndpointReference xmlns:ns01="http://schemas.xmlsoap.org/ws/2004/03/addressing">
  <ns01:Address>http://globus.my.org:8080/wsrf/services/WidgetService</ns01:Address>
  <ns01:ReferenceProperties>
    <ResourceID xmlns:ns02="http://www.w3.org/2001/XMLSchema-instance" xmlns:ns03="http://
  </ns01:ReferenceProperties>
</ns01:EndpointReference>
```

Contents of *widget:foo.xml*:

```
<doc>
  <foo xmlns="http://widgets.com"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:type="xsd:string">
    Foo Value String
  </foo>
</doc>
```

## Output and Exit Code

If the property update is successful without any output, then the program terminates with exit code 0. In the case of an error, the type of error will be displayed to *stderr* and the program will terminate with a non-0 exit code.

---

# Name

globus-wsrf-delete-property -- Delete a resource property

globus-wsrf-delete-property [OPTIONS] SERVICE-SPECIFIER PROPERTY-NAME

## Tool description

Delete a resource property from a WSRF resource.

## Command syntax

globus-wsrf-delete-property [OPTIONS]... SERVICE-SPECIFIER PROPERTY-NAME

**Table 10. Common options**

<b>-a   --anonymous</b>	Use anonymous authentication. Requires either <b>-m 'conv'</b> or transport (https) security.
<b>-d, --debug</b>	Enables debug mode. In debug mode, all SOAP messages will be displayed to stderr and full WSRF Fault messages will be displayed.
<b>-e   --eprFile FILENAME</b>	Load service EPR from FILENAME. This EPR is used to contact the WSRF service.
<b>-h   --help</b>	Displays help information about the command.
<b>-k   --key KEYNAME VALUE</b>	Set resource key in the service EPR to be named KEYNAME with VALUE as its value. This can be combined with <b>-s</b> to construct an EPR without having an xml file on hand. The <b>KEYNAME</b> is a QName string in the format <b>{namespaceURI}localPart</b> , while the <b>VALUE</b> is a literal string to place in the element. For example, the option <b>-k '{http://www.globus.org}MyKey' 128</b> would be rendered as <b>&lt;MyKey xmlns="http://www.globus.org"&gt;128&lt;/MyKey&gt;</b>
<b>-m, --securityMech TYPE</b>	Set authentication mechanism. TYPE is one of <b>msg</b> for WS-SecureMessage or <b>conv</b> for WS-SecureConversation.
<b>-p, --protection LEVEL</b>	Set message protection level. LEVEL is one of <b>sig</b> for digital signature or <b>enc</b> for encryption. The default is 'sig'.
<b>-s   --service ENDPOINT</b>	Set ENDPOINT the service URL to use. Will be composed with the <b>-k</b> parameter if present to add ReferenceProperties to the ENDPOINT
<b>-t   --timeout SECONDS</b>	Set client timeout to SECONDS.
<b>-u   --usage</b>	Print short usage message.
<b>-V   --version</b>	Show version information and exit.
<b>-v   --certKeyFiles CERTIFICATE-FILENAME KEY-FILENAME</b>	Use credentials located in CERTIFICATE-FILENAME and KEY-FILENAME. The key file must be unencrypted.
<b>-x   --proxyFilename FILENAME</b>	Use proxy credentials located in FILENAME.
<b>-z   --authorization TYPE</b>	Set authorization mode. TYPE can be <b>self</b> , <b>host</b> , <b>none</b> , or a string specifying the identity of the remote party. The default is <b>self</b> .
<b>--versions</b>	Show version information for all loaded modules and exit.

SERVICE-SPECIFIER: [-s URI [-k KEY VALUE] | -e FILENAME]

PROPERTY-NAME: [{Namespace-URI}]Property-Name

*Example:*

```
% globus-wsrf-delete-property \  
-s http://grid.example.org:8080/wsrf/services/WidgetService \  
-k "{http://www.globus.org/namespaces/2004/06/core}WidgetKey" 123 \  
"{http://widgets.com}foo"
```

## Output and Exit Code

If the property is successfully deleted, **globus-wsrf-delete-property** will not print out any output and will terminate with the exit code 0. In the case of an error, the type of error will be displayed to *stderr* and the program will terminate with a non-0 exit code.

---

## Name

globus-wsn-get-current-message -- Get the current message associated with a specified topic

globus-wsn-get-current-message [OPTIONS] SERVICE-SPECIFIER TOPIC-EXPRESSION

## Tool description

Get the current message associated with a specified topic.

## Command syntax

globus-wsn-get-current-message [OPTIONS]... SERVICE-SPECIFIER TOPIC-EXPRESSION

**Table 11. Application-specific options**

<b>-N   --namespace PREFIX=NAMESPACE-URI</b>	Create a namespace mapping in the XPATH context for the <i>PREFIX</i> string to resolve to the <i>NAMESPACE-URI</i> namespace in the Topic Expression.
<b>-D   --dialect DIALECT-URI</b>	Set the Topic Expression dialect to <i>DIALECT-URI</i> . If not specified, the dialect is chosen automatically between <i>http://docs.oasis-open.org/wsn/2004/06/TopicExpression/Simple</i> , <i>http://docs.oasis-open.org/wsn/2004/06/TopicExpression/Concrete</i> , and <i>http://docs.oasis-open.org/wsn/2004/06/TopicExpression/Full</i> based on the presence of substrings '*', '/', ' ', and '.' in the Topic Expression string.

**Table 12. Common options**

<b>-a   --anonymous</b>	Use anonymous authentication. Requires either -m 'conv' or transport (https) security.
<b>-d, --debug</b>	Enables debug mode. In debug mode, all SOAP messages will be displayed to stderr and full WSRF Fault messages will be displayed.
<b>-e   --eprFile FILENAME</b>	Load service EPR from FILENAME. This EPR is used to contact the WSRF service.
<b>-h   --help</b>	Displays help information about the command.
<b>-k   --key KEYNAME VALUE</b>	Set resource key in the service EPR to be named KEYNAME with VALUE as its value. This can be combined with -s to construct an EPR without having an xml file on hand. The KEYNAME is a QName string in the format {namespaceURI}localPart. while the VALUE is a literal string to place in the element. For example, the option -k '{http://www.globus.org}MyKey' 128 would be rendered as <MyKey xmlns="http://www.globus.org">128</MyKey>
<b>-m, --securityMech TYPE</b>	Set authentication mechanism. TYPE is one of <b>msg</b> for WS-SecureMessage or <b>conv</b> for WS-SecureConversation.
<b>-p, --protection LEVEL</b>	Set message protection level. LEVEL is one of <b>sig</b> for digital signature or <b>enc</b> for encryption. The default is 'sig'.
<b>-s   --service ENDPOINT</b>	Set ENDPOINT the service URL to use. Will be composed with the -k parameter if present to add ReferenceProperties to the ENDPOINT
<b>-t   --timeout SECONDS</b>	Set client timeout to SECONDS.
<b>-u   --usage</b>	Print short usage message.
<b>-V   --version</b>	Show version information and exit.
<b>-v   --certKeyFiles CERTIFICATE-FILENAME KEY-FILENAME</b>	Use credentials located in CERTIFICATE-FILENAME and KEY-FILENAME. The key file must be unencrypted.
<b>-x   --proxyFilename FILENAME</b>	Use proxy credentials located in FILENAME.
<b>-z   --authorization TYPE</b>	Set authorization mode. TYPE can be <b>self</b> , <b>host</b> , <b>none</b> , or a string specifying the identity of the remote party. The default is <b>self</b> .
<b>--versions</b>	Show version information for all loaded modules and exit.

SERVICE-SPECIFIER: [-s URI [-k KEY VALUE] | -e FILENAME]

TOPIC-EXPRESSION: [{Namespace-URI} | prefix ':']RootTopic[/ChildTopic]...  
 TOPIC-EXPRESSION [ '|' TOPIC-EXPRESSION]  
 RootChild or ChildTopic may contain '\*' (wildcard) and/or  
 '/' (all descendents)

*Example:*

```
% globus-wsn-get-current-message \
  -e widget.epr \
  -N wsrl=http://docs.oasis-open.org/wsrfl/2004/06/wsrfl-WS-ResourceLifetime-1.2-draft-
  'wsrl:TerminationTime'
```

```
<ns0:ResourcePropertyValueChangeNotification
  xmlns:ns0="http://docs.oasis-open.org/wsrfl/2004/06/wsrfl-WS-ResourceProperties-1.2-draft
```

```

xmlns:ns01="http://www.w3.org/2001/XMLSchema-instance"
ns01:type="ns00:ResourcePropertyValueChangeNotificationType">
  <ns00:NewValue
    ns01:type="ns00:NewValueType">
      <ns03:TerminationTime
        xmlns:ns02="http://www.w3.org/2001/XMLSchema"
        xmlns:ns03="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2"
        ns01:type="ns02:dateTime">2006-05-31T20:10:08Z</ns03:TerminationTime>
      </ns00:NewValue>
    </ns00:ResourcePropertyValueChangeNotification>
  </ns00:ResourcePropertyValueChangeNotification>

```

Contents of *widget.epr*:

```

<ns01:EndpointReference xmlns:ns01="http://schemas.xmlsoap.org/ws/2004/03/addressing">
  <ns01:Address>http://globus.my.org:8080/wsrf/services/WidgetService</ns01:Address>
  <ns01:ReferenceProperties>
    <ResourceID
      xmlns:ns02="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:ns03="http://www.w3.org/2001/XMLSchema"
      ns02:type="ns03:string">7f554f7c-efd9-11da-97a5-00096b86f788</ResourceID>
    </ns01:ReferenceProperties>
  </ns01:EndpointReference>

```

## Output and Exit Code

If the Topic exists and has a current message, **globus-wsn-get-current-message** will print the current message value to *stdout* and then terminate with the exit code 0. In the case of an error, the type of error will be displayed to *stderr* and the program will terminate with a non-0 exit code.

---

# Name

globus-wsn-pause-subscription -- Pause a WSRF notification subscription.

globus-wsn-pause-subscription [OPTIONS] SERVICE-SPECIFIER

## Tool description

Pause a WSRF notification subscription.

## Command syntax

globus-wsn-pause-subscription [OPTIONS]... SERVICE-SPECIFIER TOPIC-EXPRESSION

Table 13. Common options

<b>-a   --anonymous</b>	Use anonymous authentication. Requires either <b>-m 'conv'</b> or transport (https) security.
<b>-d, --debug</b>	Enables debug mode. In debug mode, all SOAP messages will be displayed to stderr and full WSRF Fault messages will be displayed.
<b>-e   --eprFile FILENAME</b>	Load service EPR from FILENAME. This EPR is used to contact the WSRF service.
<b>-h   --help</b>	Displays help information about the command.
<b>-k   --key KEYNAME VALUE</b>	Set resource key in the service EPR to be named KEYNAME with VALUE as its value. This can be combined with <b>-s</b> to construct an EPR without having an xml file on hand. The <b>KEYNAME</b> is a QName string in the format <b>{namespaceURI}localPart</b> , while the <b>VALUE</b> is a literal string to place in the element. For example, the option <b>-k '{http://www.globus.org}MyKey' 128</b> would be rendered as <b>&lt;MyKey xmlns="http://www.globus.org"&gt;128&lt;/MyKey&gt;</b>
<b>-m, --securityMech TYPE</b>	Set authentication mechanism. TYPE is one of <b>msg</b> for WS-SecureMessage or <b>conv</b> for WS-SecureConversation.
<b>-p, --protection LEVEL</b>	Set message protection level. LEVEL is one of <b>sig</b> for digital signature or <b>enc</b> for encryption. The default is 'sig'.
<b>-s   --service ENDPOINT</b>	Set ENDPOINT the service URL to use. Will be composed with the <b>-k</b> parameter if present to add ReferenceProperties to the ENDPOINT
<b>-t   --timeout SECONDS</b>	Set client timeout to SECONDS.
<b>-u   --usage</b>	Print short usage message.
<b>-V   --version</b>	Show version information and exit.
<b>-v   --certKeyFiles CERTIFICATE-FILENAME KEY-FILENAME</b>	Use credentials located in CERTIFICATE-FILENAME and KEY-FILENAME. The key file must be unencrypted.
<b>-x   --proxyFilename FILENAME</b>	Use proxy credentials located in FILENAME.
<b>-z   --authorization TYPE</b>	Set authorization mode. TYPE can be <b>self</b> , <b>host</b> , <b>none</b> , or a string specifying the identity of the remote party. The default is <b>self</b> .
<b>--versions</b>	Show version information for all loaded modules and exit.

SERVICE-SPECIFIER: [-s URI [-k KEY VALUE] | -e FILENAME]

*Example:*

```
% globus-wsn-pause-subscription \  
   -e subscription.epr
```

Contents of *subscription.epr*:

```
<ns00:EndpointReference  
  xmlns:ns00="http://schemas.xmlsoap.org/ws/2004/03/addressing">  
  <ns00:Address>http://globus.my.org:8080/wsrfl/services/SubscriptionManagerService</ns00:A  
  <ns00:ReferenceProperties>  
    <ns03:ResourceID  
      xmlns:ns01="http://www.w3.org/2001/XMLSchema-instance"  
      xmlns:ns02="http://www.w3.org/2001/XMLSchema"  
      xmlns:ns03="http://www.globus.org/docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotif  
      ns01:type="ns02:string">7d6430e4-f019-11da-1b9-00096b86f788</ns03:ResourceID>  
    </ns00:ReferenceProperties>  
  </ns00:EndpointReference>
```

## Output and Exit Code

If the subscription is successfully paused, **globus-wsn-pause-subscription** will terminate with the exit code 0. No further notifications should be expected on the Subscription resource until it is resumed again. In the case of an error, the type of error will be displayed to *stderr* and the program will terminate with a non-0 exit code.

---

# Name

globus-wsn-resume-subscription -- Resume a WSRF notification subscription.

globus-wsn-resume-subscription [OPTIONS] SERVICE-SPECIFIER

## Tool description

Resume a subscription.

## Command syntax

globus-wsn-resume-subscription [OPTIONS]... SERVICE-SPECIFIER TOPIC-EXPRESSION

Table 14. Common options

<b>-a   --anonymous</b>	Use anonymous authentication. Requires either <b>-m 'conv'</b> or transport (https) security.
<b>-d, --debug</b>	Enables debug mode. In debug mode, all SOAP messages will be displayed to stderr and full WSRF Fault messages will be displayed.
<b>-e   --eprFile FILENAME</b>	Load service EPR from FILENAME. This EPR is used to contact the WSRF service.
<b>-h   --help</b>	Displays help information about the command.
<b>-k   --key KEYNAME VALUE</b>	Set resource key in the service EPR to be named KEYNAME with VALUE as its value. This can be combined with <b>-s</b> to construct an EPR without having an xml file on hand. The <b>KEYNAME</b> is a QName string in the format <b>{namespaceURI}localPart</b> . while the <b>VALUE</b> is a literal string to place in the element. For example, the option <b>-k '{http://www.globus.org}MyKey' 128</b> would be rendered as <b>&lt;MyKey xmlns="http://www.globus.org"&gt;128&lt;/MyKey&gt;</b>
<b>-m, --securityMech TYPE</b>	Set authentication mechanism. TYPE is one of <b>msg</b> for WS-SecureMessage or <b>conv</b> for WS-SecureConversation.
<b>-p, --protection LEVEL</b>	Set message protection level. LEVEL is one of <b>sig</b> for digital signature or <b>enc</b> for encryption. The default is 'sig'.
<b>-s   --service ENDPOINT</b>	Set ENDPOINT the service URL to use. Will be composed with the <b>-k</b> parameter if present to add ReferenceProperties to the ENDPOINT
<b>-t   --timeout SECONDS</b>	Set client timeout to SECONDS.
<b>-u   --usage</b>	Print short usage message.
<b>-V   --version</b>	Show version information and exit.
<b>-v   --certKeyFiles CERTIFICATE-FILENAME KEY-FILENAME</b>	Use credentials located in CERTIFICATE-FILENAME and KEY-FILENAME. The key file must be unencrypted.
<b>-x   --proxyFilename FILENAME</b>	Use proxy credentials located in FILENAME.
<b>-z   --authorization TYPE</b>	Set authorization mode. TYPE can be <b>self</b> , <b>host</b> , <b>none</b> , or a string specifying the identity of the remote party. The default is <b>self</b> .
<b>--versions</b>	Show version information for all loaded modules and exit.

SERVICE-SPECIFIER: [-s URI [-k KEY VALUE] | -e FILENAME]

*Example:*

```
% globus-wsn-resume-subscription \
    -e subscription.epr
```

Contents of *subscription.epr*:

```
<ns00:EndpointReference
  xmlns:ns00="http://schemas.xmlsoap.org/ws/2004/03/addressing">
  <ns00:Address>http://globus.my.org:8080/wsrfl/services/SubscriptionManagerService</ns00:Address>
  <ns00:ReferenceProperties>
    <ns03:ResourceID
      xmlns:ns01="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:ns02="http://www.w3.org/2001/XMLSchema"
      xmlns:ns03="http://www.globus.org/docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification"
      ns01:type="ns02:string">7d6430e4-f019-11da-afb9-00096b86f788</ns03:ResourceID>
    </ns00:ReferenceProperties>
  </ns00:EndpointReference>
```

## Output and Exit Code

If the subscription is successfully resumed, **globus-wsn-resume-subscription** will terminate with the exit code 0. Notifications should again flow to the Subscription resource. In the case of an error, the type of error will be displayed to *stderr* and the program will terminate with a non-0 exit code.

---

# Name

globus-wsn-subscribe -- Subscribe for notification for a specified topic.

globus-wsn-subscribe [OPTIONS] SERVICE-SPECIFIER TOPIC-EXPRESSION

## Tool description

Subscribe for notification for a specified topic.

## Command syntax

globus-wsn-subscribe [OPTIONS]... SERVICE-SPECIFIER TOPIC-EXPRESSION

**Table 15. Application-specific options**

<b>-b   --subEpr FILENAME</b>	Save the Subscription Manager EPR in <i>FILENAME</i> . This EPR file can be used with the <code>globus-wsn-pause-subscription</code> and <code>globus-wsn-resume-subscription</code> commands
<b>-N   --namespace PREFIX=NAMESPACE-URI</b>	Create a namespace mapping in the XPATH context for the <i>PREFIX</i> string to resolve to the <i>NAMESPACE-URI</i> namespace in the Topic Expression.
<b>-D   --dialect DIALECT-URI</b>	Set the Topic Expression dialect to <i>DIALECT-URI</i> . If not specified, the dialect is chosen automatically between <code>http://docs.oasis-open.org/wsn/2004/06/TopicExpression/Simple</code> , <code>http://docs.oasis-open.org/wsn/2004/06/TopicExpression/Concrete</code> , and <code>http://docs.oasis-open.org/wsn/2004/06/TopicExpression/Full</code> based on the presence of substrings '*', '/', ' ', and ' ' in the Topic Expression string.

**Table 16. Common options**

<b>-a   --anonymous</b>	Use anonymous authentication. Requires either <b>-m 'conv'</b> or transport (https) security.
<b>-d, --debug</b>	Enables debug mode. In debug mode, all SOAP messages will be displayed to stderr and full WSRF Fault messages will be displayed.
<b>-e   --eprFile FILENAME</b>	Load service EPR from FILENAME. This EPR is used to contact the WSRF service.
<b>-h   --help</b>	Displays help information about the command.
<b>-k   --key KEYNAME VALUE</b>	Set resource key in the service EPR to be named KEYNAME with VALUE as its value. This can be combined with <b>-s</b> to construct an EPR without having an xml file on hand. The <b>KEYNAME</b> is a QName string in the format <b>{namespaceURI}localPart</b> . while the <b>VALUE</b> is a literal string to place in the element. For example, the option <b>-k '{http://www.globus.org}MyKey' 128</b> would be rendered as <b>&lt;MyKey xmlns="http://www.globus.org"&gt;128&lt;/MyKey&gt;</b>
<b>-m, --securityMech TYPE</b>	Set authentication mechanism. TYPE is one of <b>msg</b> for WS-SecureMessage or <b>conv</b> for WS-SecureConversation.
<b>-p, --protection LEVEL</b>	Set message protection level. LEVEL is one of <b>sig</b> for digital signature or <b>enc</b> for encryption. The default is 'sig'.
<b>-s   --service ENDPOINT</b>	Set ENDPOINT the service URL to use. Will be composed with the <b>-k</b> parameter if present to add ReferenceProperties to the ENDPOINT
<b>-t   --timeout SECONDS</b>	Set client timeout to SECONDS.
<b>-u   --usage</b>	Print short usage message.
<b>-V   --version</b>	Show version information and exit.
<b>-v   --certKeyFiles CERTIFICATE-FILENAME KEY-FILENAME</b>	Use credentials located in <b>CERTIFICATE-FILENAME</b> and <b>KEY-FILENAME</b> . The key file must be unencrypted.
<b>-x   --proxyFilename FILENAME</b>	Use proxy credentials located in <b>FILENAME</b> .
<b>-z   --authorization TYPE</b>	Set authorization mode. <b>TYPE</b> can be <b>self</b> , <b>host</b> , <b>none</b> , or a string specifying the identity of the remote party. The default is <b>self</b> .
<b>--versions</b>	Show version information for all loaded modules and exit.

SERVICE-SPECIFIER: [-s URI [-k KEY VALUE] | -e FILENAME]

TOPIC-EXPRESSION: [{Namespace-URI} | prefix ':']RootTopic[/ChildTopic]...  
 TOPIC-EXPRESSION [ '|' TOPIC-EXPRESSION]  
 RootChild or ChildTopic may contain '\*' (wildcard) and/or  
 '/' (all descendents)

*Example:*

```
% globus-wsn-subscribe \  

    -e counter.epr \  

    -N counter=http://www.counter.com \  

    'counter:Value'  

<ns02:Value  

    xmlns:ns00="http://www.w3.org/2001/XMLSchema-instance"  

    xmlns:ns01="http://www.w3.org/2001/XMLSchema"
```

```

    xmlns:ns02="http://counter.com" ns00:type="ns01:int">10</ns02:Value>
<ns02:Value
  xmlns:ns00="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ns01="http://www.w3.org/2001/XMLSchema"
  xmlns:ns02="http://counter.com"
  ns00:type="ns01:int">20</ns02:Value>

```

Contents of *counter.epr*:

```

<ns01:EndpointReference
  xmlns:ns01="http://schemas.xmlsoap.org/ws/2004/03/addressing">
  <ns01:Address>http://globus.my.org:8080/wsrf/services/CounterService</ns01:Address>
  <ns01:ReferenceProperties>
    <ns04:CounterKey
      xmlns:ns02="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:ns03="http://www.w3.org/2001/XMLSchema"
      xmlns:ns04="http://counter.com/service"
      ns02:type="ns03:string">1804289383</ns04:CounterKey>
    </ns01:ReferenceProperties>
  </ns01:EndpointReference>

```

## Output and Exit Code

**globus-wsn-subscribe** will print the contents of notification message to *stdout*. If the message is a ResourcePropertyValueChangedNotification message, then only the NewValue subelement will be displayed. Otherwise, the entire message will be displayed. This program will run until terminated by a signal. In the case of an error, the type of error will be displayed to *stderr* and the program will terminate with a non-0 exit code.

---

# Chapter 6. GT 4.2.1 Mapping WSDL and XML Schema to C

## 1. Introduction

This document defines the structure and format of the C bindings generated from WSDL and XSD schema in the Globus Toolkit 4.0. This version only provides mappings for the document/literal style of WSDL.

## 2. XML Namespace Mapping

In WSDL and XML Schema, XML namespaces are used to provide global resolution for types, elements and operations. In order to prevent clashes between local names when mapping to C, each target namespace can have an associated string defined that is prefixed to the types, variables, and filenames generated within that namespace. The format for the mapping is as follows:

### Figure 6.1. Definition: Namespace to Prefix Mapping Format

*namespace URI = prefix*

*namespace URI* must be a valid XML namespace.

*prefix* must be a string, conforming to the valid ANSI-C typename restrictions. As an example, the XML Schema in both [FooTypes schema](#) and [BarTypes schema](#) contains a `MyType` type definition.

For each global structure, union, type and function definition generated from the XML schema binding to C within that namespace, the defined prefix will be prepended to the defined type name.

As an example, we define the following segment of XML schema with the targetNamespace "http://foo.com/FooTypes".

#### FooTypes schema.

```
<schema xmlns:targetNamespace="http://foo.com/FooTypes">
  <complexType name="MyType">
    <sequence>
      <element name="MyInt" type="xsd:int"/>
      <element name="MyString" type="xsd:string"/>
    </sequence>
  </complexType>
</xsd:schema>
```

And another with the targetNamespace "http://bar.com/BarTypes".

#### BarTypes schema.

```
<schema xmlns:targetNamespace="http://bar.com/BarTypes">
  <complexType name="MyType">
    <sequence>
      <xsd:element name="MyQN" type="xsd:QName"/>
    </sequence>
  </complexType>
</xsd:schema>
```

```
</complexType>
</xsd:schema>
```

If the C bindings were generated without namespace to prefix mappings, the structures would look like this.

**C bindings for FooTypes schema.**

```
typedef struct MyType_s
{
    xsd_int MyInt;
    xsd_string MyString;
} MyType;
```

**C bindings for FooTypes schema.**

```
typedef struct MyType_s
{
    xsd_QName MyQN;
} MyType;
```

Although the generated bindings in C bindings for and C bindings for will be defined in different header files, there will be obvious name clashes if both are included into the same source file, or when the linker attempts to link two object files with these types defined. In order to prevent these clashes when binding to C, a mapping table must be provided:

**Figure 6.2. Mapping table for example schemas**

```
http://bar.com/BarTypes=bar_
http://foo.com/FooTypes=foo_
```

When this table is provided to the binding generator, the resulting bindings will look like the following.

**C bindings for FooTypes schema using the namespace to prefix mapping.**

```
typedef struct foo_MyType_s
{
    xsd_int MyInt;
    xsd_string MyString;
} foo_MyType;
```

**C bindings for the BarType namespace with Namespace to Prefix mapping. .**

```
typedef struct bar_MyType_s
{
    xsd_QName MyQN;
} bar_MyType;
```

The prefixes that are pre-pended to the type definitions in C bindings for using the namespace to prefix mapping. and C bindings for the BarType namespace with Namespace to Prefix mapping. prevent name clashes during compilation or object linking.

## 3. Canonicalization Rules

XML Schema allows for characters in name definitions that will cause C compilers to break. For example, an XML Schema type element definition may have the name *Foo-BarType*, but mapping this to C would result in a compiler error, since in C, - is the mathematical symbol for subtraction. We dictate the following rules when mapping names from XML schema to C:

- Hyphens: All instances of - become \_.
- Spaces: All instances of become \_.
- Language Keywords: All language keywords in C and C++ are capitalized. For example, *register* becomes *Register*.
- Attributes: Names of attributes get prefixed with \_ to prevent conflicts with element names.

## 4. Types

In the binding generation model we've chosen, each type defined in XML Schema gets a number of structures, functions and files generated for it. In each of the following sub-sections, we explain the different components of generated bindings for a XML schema type.

### 4.1. Generated Files

Each XML Schema type will generate a header file, a header file for the array of that type, and a source file. This breakdown allows us to include the generated type in other header and source files as appropriate. The format of the files is as follows:

- *Header*: `<prefix><typename>.h`
- *Header Array*: `<prefix><typename>_array.h`
- *Source*: `<prefix><typename>.c`

In the above, `<prefix>` refers to the namespace prefix mapped from the namespace for that type in the Namespace to Prefix mapping file. `<typename>` refers to the local name of the type.

As an example, the type *MyType* with namespace prefix *bar\_* will generate the files: `bar_MyType.h`, `bar_MyType_array.h`, and `bar_MyType.c`.

### 4.2. Generated Structures

Types in XML Schema are defined using *complexType* or *simpleType* elements. The structures and types generated in C for each defined schema type varies based on the content of the schema type. In general though, a typedef exists for each XML schema type, defining a type in C that maps directly to the type in XML schema. This is done for convenience and consistency with other types. The format of the typedef is the typename as a canonicalized and prefixed form of the XML Schema type name. The general format of the typedef is:

```
typedef ... <nsprefix><typename>;
```

The content of the generated type varies based on the parameters of the XML schema type. Some of the types are structs, some are just typedefs from other types, while some are more complex combinations of structs and unions. The rules for generation of the structures are described in the next sub-sections.

## 4.2.1. ComplexType Definitions

For complexType definitions, a struct is defined containing the complexType's contents. The format of the generated struct is as follows:

```
struct <nsprefix><typename>_s
{
<field1_type>[_(<o|array>)]      <field1_element>;
<field2_type>[_(<o|array>)]      <field2_element>;
...
};

typedef <nsprefix><typename>_s<nsprefix><typename>;
```

Where in the above, the field elements are expanded to:

```
<field?_type> = <type?_nsprefix><type?_localname>
<field?_element> = <element?_nsprefix><element?_localname>
```

For example, the following complexType definition in XML Schema:

```
<complexType name="Foo-BarType">
  <sequence>
    <element name="Foo" type="xsd:string"/>
    <element name="Bar" type="xsd:int"/>
  </sequence>
</complexType>
```

gets mapped to the following struct definition in C:

```
struct Foo_BarType_s
{
  xsd_string    Foo;
  xsd_int       Bar;
};

typedef  Foo_BarType_s Foo_BarType;
```

Each field element type definition of the generated struct also contains optional *\_o* or *\_array* suffixes. If an element field within a type definition contains *minOccurs="0"* and *maxOccurs="1"*, then the element is considered optional, and is given the *\_o* suffix. If the element field contains *minOccurs > 1*, then the element is considered an array, and is given the *\_array* suffix. For example, we modify the above complexType definition to this:

```
<complexType name="Foo-BarType">
  <sequence>
    <element name="Foo" type="xsd:string" minOccurs="0" maxOccurs="1"/>
    <element name="Bar" type="xsd:int" minOccurs="1" maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

The generated struct now becomes:

```
struct Foo_BarType_s
{
    xsd_string_o    Foo;
    xsd_int_array   Bar;
};

typedef Foo_BarType_s Foo_BarType;
```

For descriptions on what these optional and array types look like, see the following subsections on Optional Types and Arrays.

## 4.2.2. SimpleType Definitions

For simpleType definitions, if the simpleType contains no attribute definitions, then the typedef of the XML Schema type is generated from the base primitive type that the simpleType represents.

```
typedef <base_nsprefix><base_typename> <nsprefix><typename>;
```

For example, with the following simpleType definition:

```
<simpleType name="BazType">
    <restriction base="xsd:base64Binary"/>
</simpleType>
```

The typedef in this case would be:

```
typedef xsd_base64Binary BazType;
```

If the simpleType contains attribute definitions, then the type must be mapped to a C struct, so that the attributes can be maintained as well. In this case the struct contains a *base\_value* field, which is an instance of the primitive type of simpleType's restriction base. For example, the simpleType can be:

```
<simpleType name="FozType">
    <restriction base="xsd:base64Binary"/>
    <attribute name="Boz" type="xsd:string"/>
    <attribute name="Coz" type="xsd:int"/>
    <attribute name="Doz" type="xsd:string"/>
</simpleType>
```

This type definition gets mapped to the following C structure and typedef:

```
struct FozType_s
{
    xsd_base64Binary base_value;
    xsd_string        Boz;
    xsd_int           Coz;
    xsd_string        Doz;
};

typedef FozType_s FozType;
```

### 4.2.3. Optional Types

For each type, independent of how it is defined, a type that represents an optional instance is also defined:

```
typedef Foo_BarType * Foo_BarType_o;
```

This allows for values of instances to be optional, by either setting the value of such an instance to null, or initializing it to be non-null. This is useful for members of other types that are declared to have *minOccurs=0*.

### 4.2.4. Array Types

For each type, independent of how it is defined, a type that represents an array of that type is also defined:

```
typedef Foo_BarType_array_s
{
    struct Foo_BarType_s *   elements;
    int                     length;
    globus_xsd_type_info_t   type_info;
} Foo_BarType_array;
```

This allows for multiple values to exist as a single instance for a given member of a type. This is useful for members that have *maxOccurs > 1*. This type is defined in the `Foo_BarType_array.h` header.

Note that each of these different generated types will be defined in their associated header files: `Foo_BarType.h`, `BazType.h`, `FozType.h`.

### 4.2.5. Restrictions, Extensions and Choice

TODO

## 4.3. Generated Functions

For the generated C structures defined for a given type, a set of utility functions are also generated. These functions are:

### 4.3.1. Initialization

The following generated functions perform initialization of a generated type:

#### 4.3.1.1. Initialize Contents

```
globus_result_t
<nsprefix><typename>_init_contents(
    <typename> *                               instance)
```

#### Parameters:

- *instance* - pointer to the variable to be initialized.

**Return Value:** `globus_result_t` - a globus return value. If this initialize succeeds, the value will be `GLOBUS_SUCCESS`, otherwise an error object reference will be returned. See the globus error API for further info.

This function allows variable instances of types to be defined, and initializes the contents of those variables to null values. This is useful for local variable definitions, such as those that might be passed as input parameters to operations. For the *Foo-BarType* defined in the previous sub-section, the init contents function will be:

```
globus_result_t
Foo_BarType_init_contents(
    Foo_BarType * instance);
```

#### 4.3.1.2. Initialize Pointer

```
globus_result_t
<nsprefix><typename>_init(
    <typename> ** instance);
```

##### Parameters:

- *instance* - the reference to the pointer to be initialized

**Return Value:** `globus_result_t` - a globus return value. If this initialize succeeds, the value will be `GLOBUS_SUCCESS`, otherwise an error object reference will be returned. See the globus error API for further info.

This function allows variable instances of pointers-to-types to be allocated without using C memory allocation functions directly. This is useful for variables that must exist outside the scope where they are defined, or for optional instances or array elements. The example `_init` declaration for the *Foo-BarType* is:

```
globus_result_t
Foo_BarType_init(
    Foo_BarType ** instance);
```

#### 4.3.2. Destruction

The following generated functions perform destruction of a generated type:

##### 4.3.2.1. Destroy Contents

```
void
<nsprefix><typename>_destroy_contents(
    <typename> * instance);
```

**Parameters:** *instance* - the pointer to instance whose members are to be destroyed.

**Return value:** None

The `destroy_contents` function provides a convenient method of destruction for all the members of an instance. This function steps through the members of `<typename>` and calls the associated destruction functions for those members. An associated `init_contents` function will likely have been called previously. The example declaration is:

```
void
Foo_BarType_destroy_contents(
    Foo_BarType * instance);
```

This function can be called for locally declared variable instances that you want to be sure don't have any memory allocated members.

### 4.3.2.2. Destroy Pointer

```
void
<nsprefix><typename>_destroy(
    <typename> *    instance);
```

**Parameters:** *instance* - the pointer to be destroyed. The members of the instance pointed to are destroyed first

**Return Value:** None

This function allows a pointer-to-type instance to be deallocated conveniently. Most of the time, this will be called on an instance that was previously allocated with the associated *init* function. This function does the same thing as *destroy\_contents* function defined above, but also deallocates the memory associated with the pointer.

### 4.3.3. Duplication

The following generated functions perform duplication of a generated type:

#### 4.3.3.1. Copy Contents

```
globus_result_t
<nsprefix><typename>_copy_contents(
    <typename> *    dest,
    <typename> *    src);
```

**Parameters:**

- *dest* - the destination instance that contains the copied contents.
- *src* - the instance to copy.

**Return Value:** *globus\_result\_t* - a globus return value. If this copy succeeds, the value will be GLOBUS\_SUCCESS, otherwise an error object reference will be returned. See the globus error API for further info.

This function copies the contents of *src* to the contents of *dest* using the associated copy function for each member. The *Foo-BarType* example declaration is:

```
globus_result_t
Foo_BarType_copy_contents(
    Foo_BarType *    dest,
    Foo_BarType *    src);
```

#### 4.3.3.2. Copy Pointer

```
globus_result_t
<nsprefix><typename>_copy(
<typename> **    dest,
<typename> *    src);
```

**Parameters:**

- *dest* - the reference to the pointer to be initialized and copied to.
- *src* - the instance to copy

**Return Value:** *globus\_result\_t* - a globus return value. If this copy succeeds, the value will be GLOBUS\_SUCCESS, otherwise an error object reference will be returned. See the globus error API for further info.

This function first allocates a pointer for the new instance, and then copies the contents of *src* over to the new pointer instance. *dest* is dereferenced and set to that new pointer.

### 4.3.4. Serialization

The following generated functions perform serialization of the generated type.

#### 4.3.4.1. Serialize Contents

```
globus_result_t
<nsprefix><typename>_serialize_contents(
    xsd_QName *           element_name,
    <typename> *          instance,
    globus_soap_message_handle_t message,
    globus_xsd_element_options_t options);
```

**Parameters:**

- *element\_name* - because only the contents are being serialized, this parameter is ignored, but kept as part of the function signature for consistency. It should be NULL.
- *instance* - the instance whose contents (fields) are to be serialized
- *message* - the soap message handle to serialize the instance to.
- *options* - options that control the behavior of serialization

**Return value:** *globus\_result\_t* - a globus return value. If this serialization succeeds, the value will be GLOBUS\_SUCCESS, otherwise an error object reference will be returned. See the globus error API for further info.

This function serializes the contents of *instance* to the message handle. In this function, the *element\_name* parameter is ignored, but is included in the function declaration for consistency. The message must refer to a valid soap message handle, with optional values set in *options* to modify the behavior of the serialization.

#### 4.3.4.2. Serialize

```
globus_result_t
<nsprefix><typename>_serialize(
    xsd_QName *           element_name,
    <typename> *          instance,
    globus_soap_message_handle_t message,
    globus_xsd_element_options_t options);
```

**Parameters:**

- *element\_name* the QName of the outermost element of the serialized instance. This can be any valid QName.
- *instance* - the instance whose contents (fields) are to be serialized.
- *message* - the soap message handle to serialize the instance to.
- *options* - options that control the behavior of serialization.

**Return Value:** *globus\_result\_t* - a globus return value. If this serialization succeeds, the value will be GLOBUS\_SUCCESS, otherwise an error object reference will be returned. See the globus error API for further info.

This function serializes the instance to the message handle.

## 4.3.5. Deserialization

These generated functions perform deserialization of the generated type:

### 4.3.5.1. Deserialize Contents

```
globus_result_t
<nsprefix><typename>_deserialize_contents(
    xsd_QName *           element_name,
    <typename> *          instance,
    globus_soap_message_handle_t message,
    globus_xsd_element_options_t options);
```

**Parameters:**

- *element\_name* - because only the contents are being deserialized, this parameter is ignored, but kept as part of the function signature for consistency. It should be NULL.
- *instance* - the instance whose contents (fields) are to be deserialized. This parameter is filled in, so the previous values of the members are overwritten. A valid instance should probably only be passed in directly after being initialized with *init\_contents* or *init*. If this function succeeds, the contents of this field must be destroyed by the caller.
- *message* - the soap message handle to deserialize the instance to.
- *options* - options that control the behavior of deserialization

**Return Value:** *globus\_result\_t* - a globus return value. If this deserialization succeeds, the value will be GLOBUS\_SUCCESS, otherwise an error object reference will be returned. See the globus error API for further info.

This function deserializes the contents of the type <typename> from the message handle into the instance. In this function, the *element\_name* is ignored, but is included in the function declaration for consistency.

### 4.3.5.2. Deserialize

```
globus_result_t
<nsprefix><typename>_deserialize(
    xsd_QName *           element_name,
    <typename> *          instance,
    globus_soap_message_handle_t message,
    globus_xsd_element_options_t options);
```

**Parameters:**

- *element\_name* - this is a QName instance that should be the expected value for the outermost element of the XML serialized content for this instance. If this value does not match that outermost error, the return value will be an error object reference. The value of this field can be NULL, in which case, the outermost element can be anything.
- *instance* - the instance whose contents (fields) are to be deserialized. This parameter is filled in, so the previous values of the members are overwritten. A valid instance should probably only be passed in directly after being initialized with `init_contents` or `init`. If this function succeeds, the contents of this field must be destroyed by the caller.
- *message* - the soap message handle to deserialize the instance to.
- *options* - options that control the behavior of deserialization.

**>Return Value:** *globus\_result\_t* - a globus return value. If this deserialization succeeds, the value will be `GLOBUS_SUCCESS`, otherwise an error object reference will be returned. See the globus error API for further info.

This function deserializes the type `<typename>` from the message handle into the instance. The `element_name` is the expected value of the outermost element for the XML component of this message.

#### 4.3.5.3. Deserialize Pointer

```
globus_result_t
<nsprefix><typename>_deserialize_pointer(
    xsd_QName *           element_name,
    <typename> **        instance,
    globus_soap_message_handle_t message,
    globus_xsd_element_options_t options);
```

##### Parameters.

- *element\_name* - this is a QName instance that should be the expected value for the outermost element of the XML serialized content for this instance. If this value does not match that outermost error, the return value will be an error object reference. The value of this field can be NULL, in which case, the outermost element can be anything.
- *instance* - the reference to pointer whose contents (fields) are to be deserialized. A valid pointer is first allocated, then filled in. This field is dereferenced and set to that pointer. If this function succeeds, the instance this field points to may be NULL (signifying the serialized content for this type did not exist in the message). If it is non-null, it must be destroyed by the caller.
- *message* - the soap message handle to deserialize the instance from.
- *options* - options that control the behavior of deserialization

**Return Value:** *globus\_result\_t* - a globus return value. If this deserialization succeeds, the value will be `GLOBUS_SUCCESS`, otherwise an error object reference will be returned. See the globus error API for further info.

This function allows for deserialization of elements that are optionally supplied in the serialized form of the XML message (usually only useful for arrays and optional fields: `_o`). If the outermost element exists, the deserialization of this type takes place, and the instance pointed to will be filled in. If the outermost element doesn't exist, the instance pointed to will be set to NULL, and the function will return successfully.

#### 4.3.6. Array Functions

These generated functions perform affect arrays the generated type:

### 4.3.6.1. Array Push

```
<nsprefix><typename> *
<nsprefix><typename>_array_push(
    <typename>_array *          array);
```

**Parameters:**

- *array* - the array to expand with a new element.

**Return Value:** *instance* - a newly allocated element of type <nsprefix><type> that is appended to the end of the passed array.

This function allocates and initializes a new element and appends it to the specified array.

## 4.4. Global Type Variables

Each generated type in the C bindings includes generated global variables that provide information about that type. These global variables are useful primarily for marshalling and demarshalling of extensibility elements. The type information provided by each type's global variable allows for direct comparison at runtime of the type info to determine the actual type of an extensibility element. Also, the QName global variable is also used as a key into a registry of types maintained by the process. This allows the type lookup for extensibility elements to happen naturally. The format of the two global variables defined for each type are:

**QName.** An instance of type `xsd_QName` defining the qualified name for the type (the definition of the `xsd_QName` type is defined in the next section). It contains the XML Schema namespace and local name for the type.

```
xsd_QName <nsprefix><typename>_qname =
{
    "<Namespace of type>",
    "<local name of type>"
};
```

For example, if the `Foo-BarType` were defined in the `http://foobar` namespace, the generated QName variable would be:

```
xsd_QName Foo_BarType_qname =
{
    "http://foobar",
    "Foo-BarType"
};
```

**Type Info.** An instance of the type `globus_xsd_type_info_t` defining the functions used to perform initialization, copying, and marshalling. The definition of the `globus_xsd_type_info_t` type is (taken from `globus_xsd_type_info.h`):

```
struct globus_xsd_type_info_s
{
    xsd_QName *          type;
    globus_xsd_serialize_func_t  serialize;
    globus_xsd_deserialize_func_t  deserialize;
    globus_xsd_init_func_t  initialize;
    globus_xsd_destroy_func_t  destroy;
    globus_xsd_copy_func_t  copy;
    globus_xsd_init_contents_func_t  initialize_contents;
```

```

    globus_xsd_destroy_contents_func_t destroy_contents;
    globus_xsd_copy_contents_func_t    copy_contents;
    size_t                             type_size;
    globus_xsd_array_push_func_t       push;
    globus_xsd_type_info_t             contents_info;
    globus_xsd_type_info_t             array_info;
};

```

This is similar to virtual tables in C++, except the type information is held outside the actual type definition. The format of the type info variable for a given type is:

```

struct globus_xsd_type_info_s <nsprefix><typename>_info =
{
    &<nsprefix><typename>,
    <nsprefix><typename>_serialize_wrapper,
    <nsprefix><typename>_deserialize_wrapper,
    <nsprefix><typename>_init_wrapper,
    <nsprefix><typename>_destroy_wrapper,
    <nsprefix><typename>_copy_wrapper,
    <nsprefix><typename>_init_contents_wrapper,
    <nsprefix><typename>_destroy_contents_wrapper,
    <nsprefix><typename>_copy_contents_wrapper,
    sizeof(<nsprefix><typename>),
    <nsprefix><typename>_array_push_wrapper,
    &<nsprefix><typename>_contents_info,
    &<nsprefix><typename>_array_info
};

```

The `_wrapper` function pointers are nearly identical to the non-wrapper versions, except that `void *` is used in place of the actual type pointer to allow the function signatures to match. In other words, the wrapper form of `deserialize` is:

```

globus_result_t
<nsprefix><typename>_deserialize_wrapper(
    xsd_QName *          element,
    void *               value,
    globus_soap_message_handle_t message,
    globus_xsd_element_options_t options);

```

Following with our `Foo-BarType` example, the info variable definition would be:

```

globus_xsd_type_info_t Foo_BarType_info =
{
    &Foo_BarType_qname,
    Foo_BarType_serialize_wrapper,
    Foo_BarType_deserialize_wrapper,
    Foo_BarType_init_wrapper,
    Foo_BarType_destroy_wrapper,
    Foo_BarType_copy_wrapper,
    ...
}

```

## 4.5. Primitive Types

XML Schema defines a set of primitive types to represent different data formats. In order to maintain consistency, we define mappings to C primitive types, and include typedefs of the XSD primitives in C form. The following typedefs are defined in associated `xsd_<typename>.h` header files.

**Table 6.1. Primitive Type Bindings**

Type Name	Header File	Type Definition
any	xsd_any.h	<pre>typedef struct xsd_any_s {     globus_xsd_type_registry_t    registry;     globus_xsd_type_info_t       any_info;     xsd_QName *                  element;     void *                        value; } xsd_any;</pre>
anyAttributes	xsd_anyAttributes.h	<pre>typedef globus_hashtable_t xsd_anyAttributes;</pre>
anyType	xsd_anyType.h	<pre>typedef struct xsd_anyType_s {     globus_xsd_type_registry_t registry;     globus_xsd_type_info_t    any_info;     void *                    value; } xsd_anyType;</pre>
anyURI	xsd_anyURI.h	<pre>typedef char * xsd_anyURI</pre>
base64Binary	xsd_base64Binary.h	<pre>typedef struct {     char *    value;     size_t   length; } xsd_base64Binary;</pre>
boolean	xsd_boolean.h	<pre>typedef int xsd_boolean;</pre>
byte	xsd_byte.h	<pre>typedef char xsd_byte;</pre>
date	xsd_date.h	<pre>typedef struct tm xsd_date;</pre>
dateTime	xsd_dateTime.h	<pre>typedef struct tm xsd_dateTime;</pre>
decimal	xsd_decimal.h	<pre>typedef float xsd_decimal;</pre>
double	xsd_double.h	<pre>typedef double xsd_double;</pre>
duration	xsd_duration.h	<pre>typedef struct tm xsd_duration;</pre>
float	xsd_float.h	<pre>typedef float xsd_float;</pre>

Type Name	Header File	Type Definition
hexBinary	xsd_hexBinary.h	typedef struct { char *        value; size_t       length; } xsd_hexBinary;
ID	xsd_ID.h	typedef char * xsd_ID;
int	xsd_int.h	typedef int32_t xsd_int;
integer	xsd_integer.h	typedef BIGNUM * xsd_integer;
language	xsd_language.h	typedef char * xsd_language;
long	xsd_long.h	typedef int64_t xsd_long;
negativeInteger	xsd_negativeInteger.h	typedef BIGNUM * xsd_negativeInteger;
NCName	xsd_NCName.h	typedef char * xsd_NCName;
nonNegativeInteger	xsd_nonNegativeInteger.h	typedef BIGNUM * xsd_nonNegativeInteger;
nonPositiveInteger	xsd_nonPositiveInteger.h	typedef BIGNUM * xsd_nonPositiveInteger;
positiveInteger	xsd_positiveInteger.h	typedef BIGNUM * xsd_positiveInteger;
QName	xsd_QName.h	typedef struct { char *        Namespace; char *        local; } xsd_QName;
short	xsd_short.h	typedef int16_t xsd_short;
string	xsd_string.h	typedef char * xsd_string;
time	xsd_time.h	typedef struct tm xsd_time;
unsignedByte	xsd_unsignedByte.h	typedef unsigned char xsd_unsignedByte;
unsignedInt	xsd_unsignedInt.h	typedef uint32_t xsd_unsignedInt;
unsignedLong	xsd_unsignedLong.h	typedef uint64_t xsd_unsignedLong;
unsignedShort	xsd_unsignedShort.h	typedef uint16_t xsd_unsignedShort;

## 4.6. Elements

In XML Schema, top-level elements are declared that provide a QName useful for serializing types. While no structures or new types are generated by the bindings for elements (as they are for types), we do generate files for each element containing global variables that provide runtime information about the element. The files generated for each element are:

- *Header*: `<namespace><elementname>.h`
- *Source*: `lt;namespace><elementname>.c`

So for the following XML schema with a namespace to prefix mapping of "http://foobar=foo\_":

```
<schema xmlns:foo="http://foobar"
        xmlns:targetNamespace="http://foobar">
  <element name="Bar" type="foo:Foo-BarType"/>
</schema>
```

The files generated for element Bar would be `foo_Bar.h` and `foo_Bar.c`. The contents of element's header file include the QName and type info defined for that element. Defining these global variables for each element allows us to insert elements into the type registry as a method for runtime deserialization of unknown types (wildcards). We describe the format of these two global variables.

**QName.** An instance of type `xsd_QName` defining the qualified name for the type. It contains the XML Schema namespace and local name for the type.

```
xsd_QName <namespace><elementname>_qname =
{
  "<Namespace of element>",
  "<local name of element>"
};
```

For example, if the *Foo* element were defined in the "http://foobar" namespace, the generated QName variable would be:

```
xsd_QName foo_Bar_qname =
{
  "http://foobar"
  "Bar"
};
```

**Type Info.** An instance of the type `globus_xsd_type_info_t` defining the functions used to perform initialization, copying, and marshalling for the element. The type info for elements contains the same function pointers as those of the element's type, but it contains the QName of the element instead of the type. So the format of the type info variable for a given type is:

```
struct globus_xsd_type_info_s <namespace><elementname>_info =
{
  &lt;namespace><elementname>,
  <namespace><typename>_serialize_wrapper,
  <namespace><typename>_deserialize_wrapper,
  <namespace><typename>_init_wrapper,
  <namespace><typename>_destroy_wrapper,
```

```

<nsprefix><typename>_copy_wrapper,
<nsprefix><typename>_init_contents_wrapper,
<nsprefix><typename>_destroy_contents_wrapper,
<nsprefix><typename>_copy_contents_wrapper,
sizeof(<nsprefix><typename>),
<nsprefix><typename>_array_push_wrapper,
&&<nsprefix><typename>_contents_info,
&&<nsprefix><typename>_array_info
};

```

So the example type info global variable definition for the Bar element in `foo_Bar.h` would be:

```

struct globus_xsd_type_info_s foo_Bar_info =
{
    &foo_Bar_qname,
    foo_Foo_BarType_serialize_wrapper,
    foo_Foo_BarType_deserialize_wrapper,
    foo_Foo_BarType_init_wrapper,
    foo_Foo_BarType_destroy_wrapper,
    ...
};

```

## 5. Client Bindings

WSDL provides the operation definition as the method for message passing from client to web service. Mapping the operation to C naturally includes a stub function that will perform the operation invocation. In this bindings specification, we provide such a function definition, allowing the client-side developer to easily interact with a service.

The bindings specification also provides nonblocking stub functions for each operation as well. These are functions that allow the client to take advantage of the Globus Toolkit's nonblocking event handling architecture, and make many client invocations simultaneously.

### 5.1. Generated Files

The client-side bindings containing the stubs to allow invocation of service operations are generated in a set of source and header files, which are compiled into C static and dynamic libraries for linkage with client programs. For each service definition in WSDL, the following client interface files are generated:

- **Header:** `<service_prefix><service_name>_client.h`
- **Library:** `<libprefix><service_prefix><service_name>_client_bindings_<flavor>.<libsuffix>`

Note that for the client library, the format of the library name greatly depends on the platform being used, the flavor type compiled with the Globus Toolkit, and other user-defined parameters.

### 5.2. Client Module

The client bindings include a module definition that must be activated before client binding functions can be called, and deactivated once all client bindings functions have completed. The module name for a client is:

```
<SERVICE_NAME>_MODULE
```

The module activation and deactivation for the CounterService would look like this:

```
rc = globus_module_activate(COUNTERSERVICE_MODULE);
if(rc != GLOBUS_SUCCESS)
{
    ...
}
rc = globus_module_deactivate(COUNTERSERVICE_MODULE);
if(rc != GLOBUS_SUCCESS)
{
    ...
}
```

## 5.3. Client Handle

Each of the stub functions defined takes as its first parameter the client handle, which is an abstraction of the configuration and message properties of the service. This reduces the overall number of parameters passed to the stub, and provides abstraction and containment of configurable parameters for a service. The client handle is generated for each service definition in WSDL. The format of the handle is as follows:

```
typedef <service_prefix><service_name>_client_handle_s *
       <service_prefix><service_name>_client_handle_t;
```

Notice that the handle is actually a pointer to an internally defined struct, and as such can be set to NULL. A set of functions are also generated as part of the client bindings to manage the lifetime of the client handle.

### 5.3.1. Client Handle Initialize

```
globus_result_t
<service_prefix><service_name>_client_handle_init(
    <service_prefix><service_name>_client_handle_t *      handle,
    globus_soap_message_attr_t                            attrs,
    globus_handler_chain_t                                handlers);
```

#### Parameters:

- *handle* - the client handle to be initialized. The pointed to handle must be freed by the caller.
- *attrs* - the attributes to set on the handle. The attributes are copied to the handle, so the caller may destroy this parameter at any time after the invocation. May be NULL.
- *handlers* - a handler chain for user-defined management and control of message marshalling. The chain is copied to the client handle. May be NULL.

**Return value:** *globus\_result\_t* - a globus return value. If this deserialization succeeds, the value will be GLOBUS\_SUCCESS, otherwise an error object reference will be returned. See the globus error API for further info.

### 5.3.2. Client Handle Destroy

```
void
<service_prefix><service_name>_client_handle_destroy(
    <service_prefix><service_name>_client_handle_t handle);
```

**Parameters:**

- *handle* - the client handle to be destroyed.

**Return value:** NONE

The client handle is generally used throughout the lifetime of service invocations, and must not be destroyed until the final service invocation has completed. This happens when either the blocking call returns or the nonblocking callback is called.

## 5.4. Client Stubs

The client bindings generated from WSDL include stub functions for each operation that can be called by clients to invoke operations on services. This section defines the format of the generated bindings as well as how they are used. For each operation defined in a service, four groups of stub functions are generated. The four are: blocking, nonblocking request/response, nonblocking request, and nonblocking response. For each of these four functions, there are EPR counterparts that allow an operation to be invoked based on an EndpointReference instead of an endpoint URI. We detail the EPR counterparts at the end of this section.

### 5.4.1. Blocking Operation

*For a request-response operation*

```

globus_result_t
<portType_name>_<operation_name>(
    <service_name>_client_handle_t          handle,
    const char *                             endpoint,
    <operation_input_type> *                 <input_name>,
    <operation_output_type> **              <output_name>,
    <operation_fault_type> *                 fault_type,
    xsd_any **                               fault]);

```

*For a request-only operation*

```

globus_result_t
<portType_name>_<operation_name>(
    <service_name>_client_handle_t          handle,
    const char *                             endpoint,
    <operation_input_type> *                 <input_name>);

```

**Parameters:**

- *handle* - the client handle to use for the invocation.
- *endpoint* - a URI string that specifies the endpoint of the service.
- *<input\_name>* - the operation's input parameter defined in WSDL. This parameter has to be a pointer to *<operation\_input\_type>*, which is the type defined in WSDL by the input message part. The input parameter should already be initialized and filled in with appropriate values for marshalling.
- *<output\_name>* - the operation's output parameter defined in WSDL. This parameter has to be a referenced pointer to *<operation\_output\_type>*, which is the type defined in WSDL by the output message part. This parameter will only exist in the function if the operation is *request-response*. *One-way* operations do not have output parameters.

If the return value is `GLOBUS_SUCCESS`, this parameter will be filled in by the function based on the values returned in the response from the service, and the allocated pointer to `<operation_output_type>` must be destroyed with a call to `<operation_output_type>_destroy()`. If a non-zero value is returned by this function, the value of this parameter is undefined.

- *fault\_type* - the operation's fault type defined in WSDL. This parameter will only exist in the function declaration for *request-response* operations. One-way operations do not have faults. See the Faults section for possible values of this parameter. If the response from the service is not a fault message, the value of this parameter will be zero (`NOFAULT`). If the response from the service is a fault, the value of this parameter will be appropriate enumerated value of the fault type.
- *fault* - a reference to the extensibility element containing the deserialized fault. This parameter will only exist in the function declaration for *request-response* operations. *One-way* operations do not have faults. The value of this parameter is either `NULL` (if the response message is not a fault), or the deserialized contents of the fault message. See the Wildcards section for how to examine `xsd_any` types. If a fault message was returned in the response from the service, this parameter will be non-null, and must be freed by the caller with a call to `xsd_any_destroy()`.

**Return value:** *globus\_result\_t* - a globus return value. If this operation invocation succeeds, the value will be `GLOBUS_SUCCESS`, otherwise an error object reference will be returned. An error object can either be caused by a fault message from the service, or by a client side error in the marshalling and transport of the message. See the globus error API for further info.

The blocking function serializes the input to a soap message, sends the invocation of the operation request to the service, and waits for the response message. Once the response message is received, it deserializes it into the output parameter and returns. If the return value is `GLOBUS_SUCCESS`, the response parameter pointed to must be destroyed. If the return value is non-zero, the response may have been a fault, and can be checked with the *fault\_type* and *fault* parameters. If *fault* is non-null, it must be freed with `xsd_any_destroy` once the caller is finished with it. If the return value is non-zero, but the *fault\_type* is `NOFAULT`, then a client error occurred during message invocation.

As an example, we define the CounterService with the following operation:

```
<xsd:types>
  <xsd:element name="add" type="xsd:int"/>
  <xsd:element name="addResponse" type="xsd:int"/>
</xsd:types>

<wsdl:message name="AddInputMessage">
  <wsdl:part name="parameters" element="tns:add"/>
</wsdl:message>
<wsdl:message name="AddOutputMessage">
  <wsdl:part name="parameters" element="tns:addResponse"/>
</wsdl:message>
<wsdl:portType name="CounterPortType" wsrp:ResourceProperties="tns:CounterRP">

<wsdl:portType name="Counter">
  <wsdl:operation name="add">
    <wsdl:input message="tns:AddInputMessage"/>
    <wsdl:output message="tns:AddOutputMessage"/>
  </wsdl:operation>
</wsdl:portType>
```

The generated blocking function for the add operation is:

```
globus_result_t
Counter_add(
```

```

CounterService_client_handle_t    handle,
xsd_int *                          add,
xsd_int **                         addResponse,
Counter_fault_type_t              fault_type,
xsd_any *                           fault);

```

## 5.4.2. Nonblocking Operation

Here we define the functions for making nonblocking invocations to a web service. These functions use the globus callback event handling code to register events that trigger callbacks on completion. The events in this case are a request being sent, or a response being received. We define two callbacks to match these events.

### 5.4.2.1. Request Callback Template

```

void
(* <portType_name>_<operation_name>_request_callback_func_t) (
    <service_prefix><service_name>_client_handle_t    handle,
    void *                                             user_args,
    globus_result_t                                   result);

```

#### Parameters:

- *handle* - the client handle used to make the invocation. If this handle was only used for this invocation, it can be freed once this callback is called. Multiple invocations with the same handle will require reference counting.
- *user\_args* - a pointer containing the user arguments passed in during the register call.
- *result* - the result of the completed request. If an error occurred during marshalling or sending of the request, this result will be non-zero. Otherwise it will be GLOBUS\_SUCCESS.

**Return value:** NONE

This callback template gives the signature of the function that must be defined by the user. This function is passed as the third argument to the `<portType>_<operation_name>_register_request` function. Once the request has been sent, this callback gets called.

### 5.4.2.2. Response Callback Template

```

void
(* <portType_name>_<operation_name>_response_callback_func_t) (
    <service_name>_client_handle_t                handle,
    void *                                       user_args,
    globus_result_t                              result,
    const <operation_output_type> *              <operation_output_name>,
    <portType_name>_<operation_name>_fault_t    fault_type,
    const xsd_any *                              fault);

```

#### Parameters:

- *handle* - the client handle used to make the invocation. If this handle was only used for this invocation, it can be freed once this callback is called. Multiple invocations with the same handle will require reference counting.
- *user\_args* - a pointer containing the user arguments passed in during the register call.

- *result* - the result of the completed request. If an error occurred during marshalling or sending of the request, this result will be non-zero. Otherwise it will be `GLOBUS_SUCCESS`.
- *<operation\_output\_name>* - the output parameter of the operation filled in once the response is received and deserialized. This needs to be copied if the user wants to reference it outside of the callback's scope.
- *fault\_type* - the fault type of the fault sent back in the response. This will be `_NOFAULT` if no fault occurred.
- *fault* - a reference to the extensibility element containing the deserialized fault. The value of this parameter is either `NULL` (if the response message is not a fault), or the deserialized contents of the fault message. See the Wildcards section for how to examine `xsd_any` types. This needs to be copied if the user wants to reference it outside of the callback's scope.

**Return value:** `NONE`

This callback template gives the signature of the function that must be defined by the user. This function is passed as the third argument to the `<portType>_<operation_name>_register_request` function. Once the request has been sent, this callback gets called.

### 5.4.2.3. Nonblocking Request/Response

```

globus_result_t
<portType_name>_<operation_name>_register(
    <service_name>_client_handle_t          handle,
    <operation_input_type> *                <operation_input_name>,
    <portType_name>_<operation_name>_response_callback_func_t
                                          response_callback,
    void *                                  user_args);

```

#### Parameters:

- *handle* - The client handle to register the client operation invocation on.
- *endpoint* - The URI of the service to contact.
- *<operation\_input\_name>* - The operation input pointer that contains the structure to be serialized as the request to the operation.
- *response\_callback* - The callback to be called once the response is received and deserialized. This function callback must match the template `<portType>_<operation_name>_response_callback_func_t`.
- *user\_args* - The user arguments passed directly to the callback

**Return value:** *globus\_result\_t* - a globus return value. If this operation invocation succeeds, the value will be `GLOBUS_SUCCESS`, otherwise an error object reference will be returned. An error object can either be caused by a fault message from the service, or by a client side error in the marshalling and transport of the message. See the globus error API for further info.

The nonblocking function allows a client to register an operation invocation, so that other computation can take place while the response is being received. Once the response is received, the callback is called. In environments where the client bindings are installed with a non-threaded flavor, the user must poll for events to allow the message invocation to be processed. This is done with functions like `globus_poll()` or `globus_cond_wait()`. As an example, the asynchronous binding declaration for the `CounterService_add` operation and the response callback template is:

```

globus_result_t
CounterPortType_add_register(

```

```

CounterService_client_handle_t      handle,
const char *                        endpoint,
xsd_int *                           add,
CounterPortType_add_response_callback_t  callback,
void *                               user_args);

```

#### 5.4.2.4. Nonblocking Request

```

globus_result_t
<portType_name>_<operation_name>_register_request(
    <service_name>_client_handle_t      handle,
    const char *                        endpoint,
    <operation_input_type> *            <operation_input_name>,
    <portType_name>_<operation_name>_request_callback_func_t
                                        request_callback,
    void *                               user_args);

```

##### Parameters:

- *handle* - The client handle to register the client operation invocation on.
- *endpoint* - The URI of the service to invoke.
- *<operation\_input\_name>* - The operation input pointer that contains the structure to be serialized as the request to the operation.
- *request\_callback* - The callback to be called once the response is received and deserialized. This function callback must match the template `<portType>_<operation_name>_request_callback_func_t`.
- *user\_args* - The user arguments passed directly to the callback.

**Return value:** *globus\_result\_t* - a globus return value. If this operation invocation succeeds, the value will be `GLOBAL_SUCCESS`, otherwise an error object reference will be returned. An error object can either be caused by a fault message from the service, or by a client side error in the marshalling and transport of the message. See the globus error API for further info.

The nonblocking request function allows a client to register an operation invocation, so that other computation can take place while the request is being serialized and sent. Once the request is fully sent, the callback is called. In environments where the client bindings are installed with a non-threaded flavor, the user must poll for events to allow the message request to be processed. This is done with functions like `globus_poll()` or `globus_cond_wait()`. For the CounterService client, the declaration of the nonblocking request function for the add operation is:

```

globus_result_t
CounterPortType_add_register_request(
    CounterService_client_handle_t      handle,
    const char *                        endpoint,
    xsd_int *                           add,
    CounterPortType_add_request_callback_t  callback,
    void *                               user_args);

```

#### 5.4.2.5. Nonblocking Response

```

globus_result_t
<portType_name>_<operation_name>_register_response(
    <service_name>_client_handle_t      handle,

```

```

<portType_name>_<operation_name>_response_callback_func_t
                                response_callback,
void *                            user_args);

```

**Parameters:**

- *handle* - The client handle to register the client operation invocation on
- *response\_callback* - The callback to be called once the response is received and deserialized. This function callback must match the template <portType>\_<operation\_name>\_response\_callback\_func\_t.
- *user\_args* - The user arguments passed directly to the callback.

**Return value:** *globus\_result\_t* - a globus return value. If this operation invocation succeeds, the value will be GLOBUS\_SUCCESS, otherwise an error object reference will be returned. An error object can either be caused by a fault message from the service, or by a client side error in the marshalling and transport of the message. See the globus error API for further info.

The asynchronous response function allows a client to register a callback to be triggered when the response of a message invocation has been received and deserialized and is ready for processing. This allows other computation can take place while the response is being received. Once the response is received, the callback is called. In environments where the client bindings are installed with a non-threaded flavor, the user must poll for events to allow the message invocation to be processed. This is done with functions like `globus_poll()` or `globus_cond_wait()`. For the CounterService client, the declaration of the asynchronous response function that receives the response from the add operation is:

```

globus_result_t
CounterPortType_add_register_response(
    CounterService_client_handle_t      handle,
    CounterPortType_add_response_callback_t  callback,
    void *                                user_args);

```

### 5.4.3. EndpointReference Stubs

As mentioned previously, each of the above stub functions also has an EPR counterpart. This allows a client to invoke a service operation based on an EndpointReference (actually a C representation of it) instead of an endpoint URI. This can be useful for clients that have received an EndpointReference from a previous operation, such as a resource factory call. We only provide the stub templates for the three EPR functions that will be generated for each operation, the parameters and behavior of these functions is the same as their non-EPR counterparts.

```

globus_result_t
For a one-way operation
<porttype_name>_<operation_name>_epr(
    <service_name>_client_handle_t      handle,
    wsa_endpointreferencetype *        endpoint_reference,
    <operation_input_type> *           <input_name>);

```

```

For a request-response operation
globus_result_t
<portType_name>_<operation_name>_epr(
    <service_name>_client_handle_t      handle,
    wsa_EndpointReferenceType *        endpoint_reference,
    <operation_input_type> *           <input_name>,
    <operation_output_type> **        <output_name>,
    <operation_fault_type> *          fault_type,
    xsd_any **                        fault);

```

```
globus_result_t
  <portType_name>_<operation_name>_epr_register(
    <service_name>_client_handle_t      handle,
    wsa_EndpointReferenceType *        endpoint_reference,
    <operation_input_type> *           <operation_input_name>,
    <portType_name>_<operation_name>_response_callback_func_t
                                        response_callback,
    void *                               user_args);

globus_result_t
  <portType_name>_<operation_name>_epr_register_request(
    <service_name>_client_handle_t      handle,
    wsa_EndpointReferenceType *        endpoint_reference,
    <operation_input_type> *           <operation_input_name>,
    <portType_name>_<operation_name>_request_callback_func_t
                                        request_callback,
    void *                               user_args);
```

Notice that the *register\_response* stub function does not have an EPR counterpart, because that stub just receives the response from the service after the request has been invoked. The only difference between these templates and the non-EPR versions is in the second parameter, where a pointer to a `wsa_EndpointReferenceType` is passed in instead of an endpoint URI string of the service. In most scenarios, the user won't have to create the `endpoint_reference` by hand. Instead, the `endpoint_reference` will likely be returned from a previous service invocation.

## 6. Service Bindings

The implementation of a service consists of the parsing and operation invocation code that is often called the skeletons, as well as the service implementation itself. Because the input and output parameters are defined only by the particular service definition, and not necessarily known in advance, the parsing of all the types and subtypes of the input and output parameters must be contained with the service bindings themselves. This allows us to separate the generated code that is the service bindings from the state machine that handles transport, basic message handling (SOAP), and service invocations (the service container). The generated code for a service can then easily be contained in a module, which we describe in the following section.

### 6.1. Service Module

In C, the functionality of a service as defined by WSDL and the user's implementation is contained within a dynamic module, that gets loaded and used as needed by the service container. The service module is created from parsing/in-vo-cation code (the generated skeletons) for the service, and the actual service implementation. It gets compiled into a dynamic library to be loaded at runtime. The default name of the library is

```
lib<service_name>_<flavor>.<suffix>
```

This will get installed in a library sub-directory within the Globus Toolkit's install directory. The sub-directory will match the directory prefix of the endpoint for the service defined in the WSDL.

### 6.2. Service Source Files

The files generated for the service module from a WSDL service definition are:

- `<service_name>.h` - contains fault type definitions for each operation used by both client and service, as well as QName declarations for each operation.
- `<service_name>_internal_skeleton.h` - contains error and debugging macro definitions for the service implementation, to be used by the service implementer for debugging and returning errors.
- `<service_name>_skeleton.h` - contains the service impl function declarations that get implemented by the service implementer.
- `<service_name>_module.c` - contains the routing, marshalling, and invocation code that make up the service-side skeletons.
- `<service_name>_skeleton.c` - contains empty service implementation functions that must be filled in by the service implementer. NOTE: this file is poorly named. The "skeletons" are really contained within the `<service_name>_module.c` file, while this function contains the service implementation functions.

## 6.3. Service Implementation

A service definition in WSDL contains operations that are expected to perform some application-specific functionality. This functionality is defined by the service implementer in the service implementation functions. The functions themselves have the following template.

### 6.3.1. Service Initialization

```
globus_result_t
<service_name>_init(
globus_service_descriptor_t *    service_desc);
```

#### Parameters:

- *service\_desc* - The global service descriptor variable containing attributes about the service, such as the service's handler chain. Any fields that the implementer expects to be setup during service invocation should be initialized here.

**Return value:** *globus\_result\_t* - a globus return value. If this operation invocation succeeds, the value will be GLOBUS\_SUCCESS, otherwise an error object reference will be returned. An error object can either be caused by a fault message from the service, or by a client side error in the marshalling and transport of the message. See the globus error API for further info.

The service init function is called once when the service module is activated. This is done before any service invocations. A common use for this function is loading any operation providers for the service. An empty function is generated by default (returning GLOBUS\_SUCCESS), and should be filled in if the service implementer wishes to do something before any service invocations. An example service init function for the CounterService is:

```
globus_result_t
CounterService_init(
    globus_service_descriptor_t *    service_desc);
```

### 6.3.2. Service Finalize

```
globus_result_t
CounterService_finalize(
globus_service_descriptor_t *    service_desc);
```

#### Parameters:

- *service\_desc* - The global service descriptor variable containing attributes about the service, such as the service's handler chain. Any fields that the implementer initialized in service init should be destroyed here.

**Return value:** *globus\_result\_t* - a globus return value. If this operation invocation succeeds, the value will be GLOBUS\_SUCCESS, otherwise an error object reference will be returned. An error object can either be caused by a fault message from the service, or by a client side error in the marshalling and transport of the message. See the globus error API for further info.

The service finalize function is called once after all service invocations have completed, and the service container is shutting down. Any initialization/setup that takes place in the service init function should be cleaned up here.

### 6.3.3. Operation Implementation

```
globus_result_t
<portType_name>_<operation_name>_impl(
    globus_service_engine_t      engine,
    globus_soap_message_handle_t message,
    globus_service_descriptor_t * descriptor,
    <operation_input_type> *     <input_name>,
    <operation_output_type> *    <output_name>,
    const char **               fault_name,
    void **                     fault);
```

#### Parameters:

- *engine* - The service engine that processes the service invocations and manages the services modules. This parameter can be useful for accessing attributes on the service engine, or contacting other services.
- *message* - The soap message handle contains attributes and parameters associated with the current message request. Also, attributes associated with the message response can be set on the handle as well.
- *descriptor* - The service descriptor containing information about the service, such as the base path of the service, and the table of operation ids to function pointers for operation providers.
- *<input\_name>* - The input parameter that the client serialized and sent. This parameter has been deserialized by the service engine and is ready for processing to compute the next parameter.
- *<output\_name>* - The output parameter to be filled in by this function. The service implementer should assume that this parameter will be serialized and passed back to the client, so it must be filled in appropriately. The service skeleton code will call *<operation\_output\_type>\_destroy\_contents* on this parameter once the function returns and the output parameter is no longer needed.
- *fault\_name* - Allow a fault to be passed back to the client instead of filling in the *<output\_name>*. The fault name should be set with the *<portType\_name><operation\_name>FaultString()* macro and the fault enum value. If a fault is returned, the *<output\_name>* parameter is expected to be undefined and its contents will not be destroyed. This dereferenced string pointed to by this parameter should be set to NULL if no fault occurs.
- *fault* - The allocated and filled in fault structure. If the *fault\_name* element is non-null, this parameter should point to an allocated instance of they fault type, filled in with the values of the fault. This parameter is expected to be non-null if *fault\_name* is non-null, but will not cause an error if only the *fault\_name* is set to the fault string of the fault.

**Return value:** *globus\_result\_t* - a globus return value. If this operation invocation succeeds, the value must be GLOBUS\_SUCCESS, otherwise an error object reference must be returned. The service implementer should use either

<service\_name>\_<operation\_name>\_error() or <service\_name>\_<operation\_name>\_chain\_error() to create the error result. If the return value is not GLOBUS\_SUCCESS, a fault will be returned to the client.

The operation impl function is the meat and potatoes of the service. The service implementer should fill in this function with the code that performs the appropriate transactions for the service operation, and set the output parameter or fault parameters as appropriate. If an unrecoverable error occurs, the result returned should be non-null (GLOBUS\_SUCCESS). In this case a fault will be returned.

## 7. Faults

Each operation defined in WSDL can also have a set of associated fault types that may be returned from the service. The service implementer may choose to return faults either due to invalid processing of the input parameter or due to failures in computation for the operation. The generated bindings in C include an enum type of the valid fault types available for the operation, as well as strings of the fault typenames. The bindings also provide a mapping table from enum value to fault typename string. The enum type for a particular service operation looks like this:

```
typedef enum
{
    <PORTTYPE_NAME>_<OPERATION_NAME>_NOFAULT,
    <PORTTYPE_NAME>_<OPERATION_NAME>_UNKNOWN_FAULT,
    <PORTTYPE_NAME>_<OPERATION_NAME>_<FAULT1_NAME>,
    ...
    <PORTTYPE_NAME>_<OPERATION_NAME>_<FAULTN_NAME>
} <portType_name>_<operation_name>_fault_t;
```

This type is used in the client stubs as a parameter in which to return the fault type. The enum values can also be used by the service to access the fault typename strings which must be returned by the service impl function if a fault has occurred. In order to access the fault typename string based on the fault type enum value, the following macro is defined for each operation:

```
<portType_name>_<operation_name>FaultString(FAULT_TYPE)
```

This macro can be called with the enum value of the appropriate fault type to get the fault typename as a string as the result. As an example of the fault type generated for a service operation, we list the fault type definition for the CounterService's Destroy operation:

```
typedef enum
{
    COUNTERPORTTYPE_DESTROY_NOFAULT = 0,
    COUNTERPORTTYPE_DESTROY_UNKNOWN_FAULT = 1,
    COUNTERPORTTYPE_DESTROY_RESOURCEUNKNOWNFAULT,
    COUNTERPORTTYPE_DESTROY_RESOURCENOTDESTROYEDFAULT
} CounterPortType_Destroy_fault_t;
```

## 8. Errors

The bindings generated for each operation include macro definitions for the creation of globus\_result\_t error instances, which can returned from service implementation functions. These should be used when the operation's service implementation is unable to proceed due to invalid input parameters or some other failure specific to the service. The two macros defined and made available in the service implementation file are as follows:

## 8.1. Root Error

```
globus_result_t  
<portType_name>_<operation_name>_error(MESSAGE);
```

The MESSAGE parameter passed to this macro must be a null-terminated string containing a description of the error.

## 8.2. Non-Root Error

```
globus_result_t  
<portType_name>_<operation_name>_chain_error(RESULT, MESSAGE);
```

The RESULT parameter passed as the first argument to this macro must be a valid `globus_result_t` error reference, created by some other globus API function. The MESSAGE parameter must be a null-terminated string containing a description of the error. This macro is useful in scenarios where a globus API function is called and it returns a failure result (not equal to `GLOBUS_SUCCESS`). This allows that error reference to be chained and returned by the service implementation.

---

# Chapter 7. GT 4.2.1 Samples for C WS Core

## 1. Counter Client

The Counter Client consists of a set of client programs that can be run to interact with the CounterService by creating new counter resources, calling add on those resources, and finally destroying those resources. The reference to each resource (the EPR) is stored in a file.

The sample is a good way to get going fast with C WS-Core client programming, as the user does not have to install/deploy the CounterService—it is installed by default in GT4 containers.

- `create_count.c`<sup>1</sup> - this program invokes the createCounter operation on the CounterService and stores the resulting EPR that points to the new counter resource in a file.
- `add_count.c`<sup>2</sup> - this program reads the EPR file and invokes the add operation on the resource (of the CounterService) pointed to by the EPR.
- `destroy_count.c`<sup>3</sup> - this program reads the EPR file and destroys the resource pointed to by the EPR. Once the resource is destroyed, the EPR is no longer valid, so the file is removed.
- `Makefile.example`<sup>4</sup> - a Makefile to use for building the counter samples.

### Building the Example

#### Environment variables

`GLOBUS_LOCATION` Path where the Globus Toolkit 4.2.1 is installed.

`GLOBUS_FLAVOR_NAME` GPT flavor to build the samples with (e.g. gcc32dbg).

#### 1. Generate makefile-header

```
globus-makefile-header --flavor=$GLOBUS_FLAVOR_NAME globus_c_wsrf_sample_counter_binding
```

#### 2. Build the examples

```
make -f Makefile.example
```

---

<sup>1</sup> tutorials/counter/client/create\_count.c

<sup>2</sup> tutorials/counter/client/add\_count.c

<sup>3</sup> tutorials/counter/client/destroy\_count.c

<sup>4</sup> tutorials/counter/client/Makefile.example

---

# Chapter 8. Debugging

## 1. Environment variables

Besides the standard debugging tools available on your platform for C programs, the C WS-Core has a number of environment variables that can be set to debug or trace program execution of the service container. The useful environment variables that can be set are:

- GLOBUS\_SERVICE\_ENGINE\_DEBUG - useful for tracing execution of the service engine. The possible values this variable can have are:
  - DEBUG - show debug messages about execution of the engine.
  - INFO - show information regarding service invocations.
  - TRACE - show entry and exit points of functions for the service engine.
  - ERROR - show error occurring during service invocation.
  - ALL - all the above levels joined together.

## 2. Logging

As of 4.2.1, the Globus Toolkit provides system administration logs that are [CEDPs best practices](http://cedps.net/index.php/LoggingBestPractices)<sup>1</sup> compliant.

To enable CEDPS logging, pass the `-log PATH` parameter to the **globus-wsc-container** program.

For more details on the CEDPS Logging format, including descriptions of reserved name-value pairs, see <http://cedps.net/index.php/LoggingBestPractices>:

### 2.1. Sample log file

The [sample log file](http://www.globus.org/toolkit/docs/4.2/4.2.1/common/cwscore/sample-container-log.txt)<sup>2</sup> contains many log entries for various scenarios in the C WS container.

---

<sup>1</sup> <http://cedps.net/index.php/LoggingBestPractices>

<sup>2</sup> <http://www.globus.org/toolkit/docs/4.2/4.2.1/common/cwscore/sample-container-log.txt>

---

# Chapter 9. Troubleshooting

This is a new component. If you're having trouble, [please let us know](#)<sup>1</sup> [fixme link to support page].

For a list of common errors in GT, see [Error Codes](#).

---

<sup>1</sup> /toolkit/support.html

# 1. C WS Core Errors

**Table 9.1. C WS Core Errors**

Error Code	Definition	Possible Solutions
<p>globus_soap_message_module: Failed sending request http://wid-gets.com/WidgetPortType/createWidgetRequest. globus_xio: Unable to connect to grid.example.org:8080 globus_xio: System error in connect: Connection refused globus_xio: A system call failed: Connection refused</p>	<p>Unable to contact service container</p>	<p>Check that the service endpoint refers to a running container.</p>
<p>globus_soap_message_module: Failed sending request http://wid-gets.com/WidgetPortType/createWidgetRequest. globus_xio_gsi: gss_init_sec_context failed. GSS Major Status: Unexpected Gatekeeper or Service Name globus_gsi_gss-api: Authorization denied: The name of the remote entity (/C=US/O=Globus Alliance/OU=Service/CN=host/grid.example.org), and the expected name for the remote entity (/C=US/O=Globus Alliance/OU=Service/CN=host/cloud.example.org) do not match</p>	<p>Service is not running with the expected security credential.</p>	<p>Verify that the service credential being presented by the service (first parenthesized name) is a reasonable certificate name for the service. If so, set the GLOBUS_SOAP_MESSAGE_PEER_IDENTITY_KEY attribute on the soap message handle to that identity. For most command-line wsrfl tools, this can be done by passing it as an argument to the -z command-line parameter.</p>
<p>globus_soap_message_module: SOAP Fault Fault code: Client Fault string: globus_service_engine_module: Failed to find operation: {XXXX}YYYY for service: {ZZZZ}BBBB</p>	<p>The service port type {ZZZZ}BBBB does not contain a {XXXX}YYYY operation.</p>	<p>Verify that the client bindings are built from the same WSDL and XML Schema documents as the service.</p>
<p>globus_soap_message_module: Failed receiving response http://wid-gets.com/WidgetPortType/createWidgetResponse. ws_addressing: Addressing header is a draft version of WS Addressing: "http://schemas.xmlsoap.org/ws/2004/03/addressing". This could be a GT version mismatch, client is GT 4.2.x and response is from GT 4.0.x server</p>	<p>The service is running on a container which is using a draft version of the WS-Addressing specification. This was used by GT 4.0.x</p>	<p>Update the service to work with GT 4.2.x or compile your client with GT 4.0.x libraries.</p>
<p>globus_soap_message_module: Failed sending request http://wid-gets.com/WidgetPortType/createWidgetRequest. globus_xio: The GSI XIO driver failed to establish a secure connection. The failure occurred during a handshake read. globus_xio: An end of file occurred</p>	<p>The service container either did not support SSL authentication, or the service container did not trust the client certificate</p>	<p>Consult the service administrator to verify that the service container supports SSL and that your certificate is issued by a certificate authority trusted by the service.</p>

---

# Chapter 10. Related Documentation

None at present.

---

# Glossary

## F

flavor Pre-OGSI Globus description term that uniquely encompasses Machine Architecture, OS, Compiler and other attributes into a single term, for example: gcc32dbgpthr for a threaded Linux debug distribution.

## S

SOAP SOAP provides a standard, extensible, composable framework for packaging and exchanging XML messages between a service provider and a service requester. SOAP is independent of the underlying transport protocol, but is most commonly carried on HTTP. See the [SOAP specifications](#)<sup>13</sup> for details.

## T

transport-level security Uses transport-level security (TLS) mechanisms.

## W

Web Services Description Language (WSDL) WSDL is an XML document for describing Web services. Standardized binding conventions define how to use WSDL in conjunction with SOAP and other messaging substrates. WSDL interfaces can be compiled to generate proxy code that constructs messages and manages communications on behalf of the client application. The proxy automatically maps the XML message structures into native language objects that can be directly manipulated by the application. The proxy frees the developer from having to understand and manipulate XML. See the [WSDL 1.1 specification](#)<sup>15</sup> for details.

Web Services Resource Framework (WSRF) Web Services Resource Framework (WSRF) is a specification that extends web services for grid applications by giving them the ability to retain state information while at the same time retaining statelessness (using resources). The combination of a web service and a resource is referred to as a WS-Resource. WSRF is a collection of different specifications that manage WS-Resources.

This framework comprises mechanisms to describe views on the state (WS-ResourceProperties), to support management of the state through properties associated with the Web service (WS-ResourceLifetime), to describe how these mechanisms are extensible to groups of Web services (WS-ServiceGroup), and to deal with faults (WS-BaseFaults).

For more information, go to: <http://www.globus.org/wsrf/> and [OASIS Web Services Notification \(WSRF\) TC](#)<sup>19</sup>.

---

<sup>13</sup> <http://www.w3.org/TR/soap/>

<sup>15</sup> <http://www.w3.org/TR/wsdl>

<sup>19</sup> [http://www.oasis-open.org/committees/tc\\_home.php?wg\\_abbrev=wsrf](http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf)

---

# Index

## B

BlogService  
  implementing, 13  
  writing clients for, 5

## C

client stubs  
  using asynchronous, 4  
clients  
  writing clients for the BlogService, 5  
containers  
  globus-wsc-container, 26  
  starting C standalone container, 26

## D

debugging, 92

## E

errors, 94

## L

logging, 92

## N

notification  
  getting current message associated with specific topic,  
  51  
  globus-wsn-get-current-message, 51  
  globus-wsn-pause-subscription, 54  
  globus-wsn-resume-subscription, 56  
  globus-wsn-subscribe, 58  
  pausing subscription, 54  
  resuming subscription, 56  
  subscribing, 58

## R

resource  
  destroying, 32  
  globus-wsrf-destroy, 32  
  globus-wsrf-query, 36  
  globus-wsrf-set-termination-time, 34  
  querying resource properties, 36  
  setting the scheduled termination time, 34  
resource properties  
  deleting, 49  
  getting multiple values, 41  
  getting value, 39  
  globus-wsrf-delete-property, 49

globus-wsrf-get-properties, 41  
globus-wsrf-get-property, 39  
globus-wsrf-insert-property, 43  
globus-wsrf-update-property, 46  
inserting a value, 43  
querying, 36  
updating a value, 46

## S

samples  
  counter client, 91  
services  
  implementing the BlogService, 13  
stubs  
  generating C bindings from WSDL schema files, 28  
  globus-wsrf-cgen, 28  
subscribing, 58  
  pausing, 54  
  resuming, 56

## W

wildcards  
  using, 3