

Mapping WSDL and XSD Schema to C Globus Toolkit 4.0

Table of Contents

Table of Contents.....	1
Introduction	2
XML Namespace Mapping	2
Canonicalization Rules.....	5
Types	5
Generated Files	5
Generated Structures.....	6
ComplexType Definitions.....	6
SimpleType Definitions.....	8
Optional Types	9
Array Types	10
Restrictions, Extensions and Choice	10
Generated Functions	10
Initialization	10
Destruction	12
Duplication.....	13
Serialization	14
Deserialization.....	15
Global Type Variables.....	18
Primitive Types.....	22
Elements.....	27
Client Bindings	29
Generated Files	30
Client Module	30
Client Handle.....	31
Client Stubs	33
Blocking Operation	33
Asynchronous Operation	36
Asynchronous Request/Response	38
Asynchronous Request.....	39
Asynchronous Response	40
EndpointReference Stubs	41
Service Bindings	42
Service Module	43
Service Source Files.....	43
Service Implementation	43
Service Init	44
Service Finalize	45

Operation Init.....	46
Operation Impl.....	47
Faults	49
Errors	50

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 document will be of interest if you want to use the C bindings to write web services and clients. It contains information as to the structure and formatting of the types, client bindings, and service bindings generated in C, as well as how we address a number of bindings related issues with mapping an abstract schema language like WSDL and XSD to the C programming language. This document's scope only includes the generated code that makes up the C bindings as public interfaces to the user. As such the tools used for code generation, and the exact implementation details of API functions are not discussed here. This is NOT a normative reference. The normative reference for the C bindings are the bindings generation templates themselves. This version of the document only provides mappings for the document/literal style of WSDL, so it is assumed that there will only be at most one input and output element for each operation.

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:

Definition: Namespace to Prefix Mapping Format

<code><namespace URI> = <prefix></code>

<namespace URI> must be a valid XML namespace.

<prefix> needs to be a string, conforming to the valid ANSI-C typename restrictions. As an example, the XML Schema in both Figure 1 and Figure 2 contains the 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 pre-pended to the definition.

As an example, we define the following segment of XML schema:

```
<xsd:schema ... xmlns:targetNamespace="http://foo.com/FooTypes">
  ...

  <complexType name="MyType">
    <sequence>
      <xsd:element name="MyInt" type="xsd:int"/>
      <xsd:element name="MyString" type="xsd:string"/>
    </sequence>
  </complexType>

  ...
</xsd:schema>
```

Figure 1: Example XML Schema type definition with a target namespace

```
<xsd:schema ... xmlns:targetNamespace="http://bar.com/BarTypes">
  ...

  <complexType name="MyType">
    <sequence>
      <xsd:element name="MyQN" type="xsd:QName"/>
    </sequence>
  </complexType>

  ...
</xsd:schema>
```

Figure 2: Example XML Schema type definition with a different target namespace

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

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

Figure 3: Snippet of C binding from XSD Schema in Figure 1

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

Figure 4: Snippet of C binding from XSD Schema in Figure 2

Although the generated bindings in Figure 3 and Figure 4 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:

```
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 this:

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

Figure 5: C bindings for the Foo Namespace with Namespace to Prefix mapping

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

Figure 6: C bindings for the BarType namespace with Namespace to Prefix mapping

The prefixes that are pre-pended to the type definitions in Figure 5 and Figure 6 prevent name clashes during compilation or object linking.

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 '_'
- Restricted Names: All restricted names in C and C++ are capitalized. i.e. *register* becomes *Register*
- Attributes: Names of attribute definitions in types get prefixed with '_' to prevent conflicts with other elements of the same name.

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.

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

In the above, <nsprefix> 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

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.

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>[_(olarray)]    <field1_element>;
    <field2_type>[_(olarray)]    <field2_element>;
    ...
}

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

```

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

```

<field?_type> = <type?_namespace><type?_localname>
<field?_element> = <element?_namespace><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](#).

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;
```

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.

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;  
} 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, FozType.

Restrictions, Extensions and Choice

Generated Functions

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

Initialization

The following generated functions perform initialization of a generated type:

Initialize Contents

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

Parameters:

- instance – the 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);
```

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);
```

Destruction

The following generated functions perform destruction of a generated type:

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.

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.

Duplication

The following generated functions perform duplication of a generated type:

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 from

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);
```

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 pointer to copy from

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.

Serialization

The following generated functions perform serialization of the generated type:

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.

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.

Deserialization

These generated functions perform deserialization of the generated type:

Deserailize 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.

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.

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 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 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.

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 <namespace><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 global 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:

```
<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,  
    ...  
}
```

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 same initialization and serialization functions we defined in the previous section exist for each of the XSD primitives defined below. The following typedefs are defined in associated `xsd_<typename>.h` header files:

Type: any
<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>
Header File: <code>xsd_any.h</code>

Type: anyAttributes
<pre>typedef globus_hashtable_t xsd_anyAttributes;</pre>
Header File: <code>xsd_anyAttributes.h</code>

Type: anyType
<pre>typedef struct xsd_anyType_s { globus_xsd_type_registry_t registry; globus_xsd_type_info_t any_info; void * value; } xsd_anyType;</pre>
Header File: <code>xsd_anyType.h</code>

Type: anyURI
<pre>typedef char * xsd_anyURI;</pre>
Header File: <code>xsd_anyURI.h</code>

Type: **base64Binary**

```
typedef struct
{
    char *    value;
    size_t   length;
} xsd_base64Binary;
```

Header File: xsd_base64Binary.h

Base64 is a frequently used type for converting generic (non-ascii) byte arrays into content that fits within the XML utf-8 encoding. The value field of this type can be set to any byte array and the length to the length of that byte array. The conversion to base64 happens when the type is serialized.

Type: **boolean**

```
typedef int    xsd_boolean;
```

Header File: xsd_boolean.h

Type: **byte**

```
typedef char   xsd_byte;
```

Header File: xsd_byte.h

Type: **date**

```
typedef struct tm    xsd_date;
```

Header File: xsd_date.h

The struct tm type is a frequently used type for Unix systems to hold date and time information. There are a number of standard Unix functions for creating and manipulating instances of struct tm.

Type: **dateTime**

```
typedef struct tm    xsd_dateTime;
```

Header File: xsd_dateTime.h

Type: **decimal**

```
typedef float xsd_decimal;
```

Header File: xsd_decimal.h

Type: **double**

```
typedef double xsd_double;
```

Header File: xsd_double.h

Type: **duration**

```
typedef struct tm xsd_duration;
```

Header File: xsd_duration.h

Type: **float**

```
typedef float xsd_float;
```

Header File: xsd_float.h

Type: **hexBinary**

```
typedef struct
{
    char * value;
    size_t length;
} xsd_hexBinary;
```

Header File: xsd_hexBinary.h

Type: **ID**

```
typedef char * xsd_ID;
```

Header File: xsd_ID.h

Type: **int**

```
typedef int32_t xsd_int;
```

Header File: xsd_int.h

Type: integer
<pre>typedef BIGNUM * xsd_integer;</pre>
Header File: xsd_integer.h

For unbounded integers, the BIGNUM type from the openssl package is used. This allows for infinitely large numbers (or at least limited by the memory on the system).

Type: language
<pre>typedef char * xsd_language;</pre>
Header File: xsd_language.h

Type: long
<pre>typedef int64_t xsd_long;</pre>
Header File: xsd_long.h

Type: negativeInteger
<pre>typedef BIGNUM * xsd_negativeInteger;</pre>
Header File: xsd_negativeInteger.h

Type: NCName
<pre>typedef char * xsd_NCName;</pre>
Header File: xsd_NCName.h

Type: nonNegativeInteger
<pre>typedef BIGNUM * xsd_nonNegativeInteger;</pre>
Header File: xsd_nonNegativeInteger.h

Type: nonPositiveInteger
<pre>typedef BIGNUM * xsd_nonPositiveInteger;</pre>

Header File: xsd_nonPositiveInteger.h

Type: **positiveInteger**

```
typedef BIGNUM * xsd_positiveInteger;
```

Header File: xsd_positiveInteger.h

Type: **QName**

```
typedef struct  
{  
    char *    Namespace;  
    char *    local;  
} xsd_QName;
```

Header File: xsd_QName.h

Type: **short**

```
typedef int16_t    xsd_short;
```

Header File: xsd_short.h

Type: **string**

```
typedef char *    xsd_string;
```

Header File: xsd_string.h

Type: **time**

```
typedef struct tm    xsd_time;
```

Header File: xsd_time.h

Type: **unsignedByte**

```
typedef unsigned char    xsd_unsignedByte;
```

Header File: xsd_unsignedByte.h

Type: unsignedInt
typedef uint32_t xsd_unsignedInt;
Header File: xsd_unsignedInt.h

Type: unsignedLong
typedef uint64_t xsd_unsignedLong;
Header File: xsd_unsignedLong.h

Type: unsignedShort
typedef uint16_t xsd_unsignedShort;
Header File: xsd_unsignedShort.h

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:** <nsprefix><elementname>.h
- **Source:** <nsprefix><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"/>
```

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    <nsprefix><elementname>_info =
{
    &<nsprefix><elementname>,
    <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
};

```

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,
    ...
};

```

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 asynchronous stub functions for each operation as well. These are functions that allow the client to take

advantage of the Globus Toolkit's asynchronous event handling architecture, and make many client invocations asynchronously.

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. Example headers and libraries from the CounterService would be:

- Header: CounterService_client.h
- Library: libCounterService_client_bindings_gcc32dbg.so

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
```

So 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)
{
    ...
}
```

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:

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.

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 same client handle can be used for multiple service invocations (stub calls), but only sequentially. The client handle maintains state about the connection for the receiving the response from the service, so if using any of the asynchronous stubs, then you should use a new client handle for each operation invocation. For asynchronous request/response stubs, the same client handle should be passed to the *register_response* stub that was used for the request. The client handle must not be freed until either the blocking call returns (in the blocking case) or the asynchronous call's callback is called (for the asynchronous case).

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, asynchronous request/response, asynchronous request, and asynchronous 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.

Blocking 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           ]);
```

Parameters:

- handle – the client handle to use for the invocation
- endpoint – This is 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>` [OPTIONAL] – 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` [OPTIONAL] – 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` [OPTIONAL] – 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,
    const char *                       endpoint,
    xsd_int *                           add,
    xsd_int **                          addResponse,
    Counter_fault_type_t               fault_type,
    xsd_any *                            fault);

```

Asynchronous Operation

Here we define the functions for making asynchronous 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.

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.

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 the fault type was a

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 function. Once the response has been received and deserialized, this callback gets called.

Asynchronous Request/Response

```
globus_result_t
<portType_name>_<operation_name>_register(
    <service_name>_client_handle_t      handle,
    const char *                        endpoint,
    <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 asynchronous 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);

```

Asynchronous 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 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 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 asynchronous 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);
```

Asynchronous 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:

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

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
<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.

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.

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/invoke 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.

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.

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:

Service Init

```
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);
```

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.

Operation Init

```
globus_result_t
<portType_name>_<operation_name>_init(
    globus_service_engine_t      engine,
    globus_soap_message_handle_t message,
    <operation_input_type> *     <input_name>);
```

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.
- <input_name> - The input parameter to be initialized before deserialization into this field takes place. This allows fine-grained control of deserialization of certain fields in the input parameter such as wildcards.

Return value: globus_result_t – a globus return value. If this operation init succeeds, the value returned 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 init function allows the service implementer to perform actions before deserialization of the soap message takes place. An empty function that returns GLOBUS_SUCCESS is generated by default. If the user wishes to override the default, he should replace that empty function with his own implementation.

Operation Impl

```
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 the 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.

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;
```

Errors

The bindings generated for each operation include macro definitions for the creation of `globus_result_t` error instances, which can be 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:

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.

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.