

# globus common Reference Manual

## 14.5

Generated by Doxygen 1.4.7

Thu Feb 2 06:14:52 2012

# Contents

<b>1</b>	<b>globus common Module Index</b>	<b>1</b>
<b>2</b>	<b>globus common Data Structure Index</b>	<b>1</b>
<b>3</b>	<b>globus common Page Index</b>	<b>2</b>
<b>4</b>	<b>globus common Module Documentation</b>	<b>2</b>
<b>5</b>	<b>globus common Data Structure Documentation</b>	<b>52</b>
<b>6</b>	<b>globus common Page Documentation</b>	<b>55</b>

## 1 globus common Module Index

### 1.1 globus common Modules

Here is a list of all modules:

<b>Globus Callback</b>	<b>2</b>
<b>Globus Callback API</b>	<b>4</b>
<b>Globus Callback Spaces</b>	<b>12</b>
<b>Globus Callback Signal Handling</b>	<b>16</b>
<b>Globus Error API</b>	<b>22</b>
<b>Globus Errno Error API</b>	<b>18</b>
<b>Error Construction</b>	<b>18</b>
<b>Error Data Accessors and Modifiers</b>	<b>20</b>
<b>Error Handling Helpers</b>	<b>21</b>
<b>Globus Generic Error API</b>	<b>23</b>
<b>Error Construction</b>	<b>23</b>
<b>Error Data Accessors and Modifiers</b>	<b>26</b>
<b>Error Handling Helpers</b>	<b>29</b>
<b>Threading</b>	<b>31</b>
<b>Mutual Exclusion</b>	<b>36</b>
<b>Condition Variables</b>	<b>41</b>
<b>Thread-Specific Storage</b>	<b>45</b>
<b>One-time execution</b>	<b>47</b>

## 2 globus common Data Structure Index

### 2.1 globus common Data Structures

Here are the data structures with brief descriptions:

<b>globus_cond_t (Condition variable )</b>	<b>52</b>
<b>globus_condattr_t (Condition variable attribute )</b>	<b>52</b>
<b>globus_mutex_t (Mutex )</b>	<b>52</b>
<b>globus_mutexattr_t (Mutex attribute )</b>	<b>52</b>
<b>globus_rmutex_t (Recursive Mutex )</b>	<b>53</b>
<b>globus_thread_key_t (Thread-specific data key )</b>	<b>53</b>
<b>globus_thread_once_t (Thread once structure )</b>	<b>53</b>
<b>globus_thread_t (Thread ID )</b>	<b>53</b>
<b>globus_threadattr_t (Thread attributes )</b>	<b>53</b>
<b>globus_url_t (Parsed URLs )</b>	<b>54</b>

## 3 globus common Page Index

### 3.1 globus common Related Pages

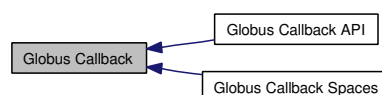
Here is a list of all related documentation pages:

<b>Deprecated List</b>	<b>55</b>
------------------------	-----------

## 4 globus common Module Documentation

### 4.1 Globus Callback

Collaboration diagram for Globus Callback:



#### Modules

- Globus Callback API
- Globus Callback Spaces

## Defines

- **#define GLOBUS\_CALLBACK\_MODULE**
- **#define GLOBUS\_POLL\_MODULE**

## Typedefs

- **typedef int globus\_callback\_handle\_t**
- **typedef int globus\_callback\_space\_t**
- **typedef globus\_l\_callback\_space\_attr\_s \* globus\_callback\_space\_attr\_t**

## Enumerations

- **enum globus\_callback\_error\_type\_t {**  
    **GLOBUS\_CALLBACK\_ERROR\_INVALID\_CALLBACK\_HANDLE = 1024,**  
    **GLOBUS\_CALLBACK\_ERROR\_INVALID\_SPACE,**  
    **GLOBUS\_CALLBACK\_ERROR\_MEMORY\_ALLOC,**  
    **GLOBUS\_CALLBACK\_ERROR\_INVALID\_ARGUMENT,**  
    **GLOBUS\_CALLBACK\_ERROR\_ALREADY\_CANCELED,**  
    **GLOBUS\_CALLBACK\_ERROR\_NO\_ACTIVE\_CALLBACK }**

### 4.1.1 Detailed Description

### 4.1.2 Define Documentation

#### 4.1.2.1 #define GLOBUS\_CALLBACK\_MODULE

Module descriptor.

Module descriptor for for globus\_callback module. Must be activated before any of the following api is called.

Note: You would not normally activate this module directly. Activating the GLOBUS\_COMMON\_MODULE will in turn activate this also.

#### 4.1.2.2 #define GLOBUS\_POLL\_MODULE

Module descriptor.

## Deprecated

Backward compatible name

### 4.1.3 Typedef Documentation

#### 4.1.3.1 typedef int globus\_callback\_handle\_t

Periodic callback handle.

This handle can be copied or compared, and represented as NULL with GLOBUS\_NULL\_HANDLE

#### 4.1.3.2 typedef int globus\_callback\_space\_t

Callback space handle.

This handle can be copied or compared and represented as NULL with GLOBUS\_NULL\_HANDLE

#### 4.1.3.3 typedef struct globus\_l\_callback\_space\_attr\_s\* globus\_callback\_space\_attr\_t

Callback space attribute.

This handle can be copied and represented as NULL with GLOBUS\_NULL

### 4.1.4 Enumeration Type Documentation

#### 4.1.4.1 enum globus\_callback\_error\_type\_t

Error types.

Possible error types returned by the api in this module. You can use the error API to check results against these types.

See also:

**Error Handling Helpers** (p. 29)

Enumerator:

**GLOBUS\_CALLBACK\_ERROR\_INVALID\_CALLBACK\_HANDLE** The callback handle is not valid or it has already been destroyed.

**GLOBUS\_CALLBACK\_ERROR\_INVALID\_SPACE** The space handle is not valid or it has already been destroyed.

**GLOBUS\_CALLBACK\_ERROR\_MEMORY\_ALLOC** Could not allocate memory for an internal structure.

**GLOBUS\_CALLBACK\_ERROR\_INVALID\_ARGUMENT** One of the arguments is NULL or out of range.

**GLOBUS\_CALLBACK\_ERROR\_ALREADY\_CANCELED** Attempt to unregister callback again.

**GLOBUS\_CALLBACK\_ERROR\_NO\_ACTIVE\_CALLBACK** Attempt to retrieve info about a callback not in callers's stack.

## 4.2 Globus Callback API

Collaboration diagram for Globus Callback API:



### Convenience Macros

- #define globus\_callback\_poll(a)
- #define globus\_poll\_blocking()
- #define globus\_poll\_nonblocking()
- #define globus\_poll()
- #define globus\_signal\_poll()
- #define globus\_callback\_register\_onehot(callback\_handle,delay\_time,callback\_func,callback\_user\_arg)

- **#define** globus\_callback\_register\_periodic(callback\_handle,delay\_time,period,callback\_func,callback\_user\_arg)
- **#define** globus\_callback\_register\_signal\_handler(signum,persist,callback\_func,callback\_user\_arg)

## Callback Prototypes

- **typedef void(\*) globus\_callback\_func\_t (void \*user\_arg)**

## Oneshot Callbacks

- **globus\_result\_t globus\_callback\_space\_register\_oneshot (globus\_callback\_handle\_t \*callback\_handle, const globus\_reftime\_t \*delay\_time, globus\_callback\_func\_t callback\_func, void \*callback\_user\_arg, globus\_callback\_space\_t space)**

## Periodic Callbacks

- **globus\_result\_t globus\_callback\_space\_register\_periodic (globus\_callback\_handle\_t \*callback\_handle, const globus\_reftime\_t \*delay\_time, const globus\_reftime\_t \*period, globus\_callback\_func\_t callback\_func, void \*callback\_user\_arg, globus\_callback\_space\_t space)**
- **globus\_result\_t globus\_callback\_unregister (globus\_callback\_handle\_t callback\_handle, globus\_callback\_func\_t unregister\_callback, void \*unreg\_arg, globus\_bool\_t \*active)**
- **globus\_result\_t globus\_callback\_adjust\_oneshot (globus\_callback\_handle\_t callback\_handle, const globus\_reftime\_t \*new\_delay)**
- **globus\_result\_t globus\_callback\_adjust\_period (globus\_callback\_handle\_t callback\_handle, const globus\_reftime\_t \*new\_period)**

## Callback Polling

- **void globus\_callback\_space\_poll (const globus\_abstime\_t \*timestop, globus\_callback\_space\_t space)**
- **void globus\_callback\_signal\_poll ()**

## Miscellaneous

- **globus\_bool\_t globus\_callback\_get\_timeout (globus\_reftime\_t \*time\_left)**
- **globus\_bool\_t globus\_callback\_has\_time\_expired ()**
- **globus\_bool\_t globus\_callback\_was\_restarted ()**

### 4.2.1 Detailed Description

### 4.2.2 Define Documentation

#### 4.2.2.1 #define globus\_callback\_poll(a)

Poll the global callback space.

Specifies the global space for **globus\_callback\_space\_poll()** (p. 10). argument is the timeout

See also:

**globus\_callback\_space\_poll()** (p. 10)

#### 4.2.2.2 #define globus\_poll\_blocking()

Blocking poll of the global callback space.

Specifies that **globus\_callback\_space\_poll()** (p. 10) should poll on the global space with an infinite timeout

See also:

**globus\_callback\_space\_poll()** (p. 10)

#### 4.2.2.3 #define globus\_poll\_nonblocking()

Nonblocking poll of the global callback space.

Specifies that **globus\_callback\_space\_poll()** (p. 10) should poll on the global space with an immediate timeout

See also:

**globus\_callback\_space\_poll()** (p. 10)

#### 4.2.2.4 #define globus\_poll()

Nonblocking poll of the global callback space.

Specifies that **globus\_callback\_space\_poll()** (p. 10) should poll on the global space with an immediate timeout

See also:

**globus\_callback\_space\_poll()** (p. 10)

#### 4.2.2.5 #define globus\_signal\_poll()

Wake up callback polling thread.

Counterpart to **globus\_poll()** (p. 6).

See also:

**globus\_callback\_signal\_poll()** (p. 11)

#### 4.2.2.6 #define globus\_callback\_register\_oneshot(callback\_handle, delay\_time, callback\_func, callback\_user\_arg)

Register a oneshot function in the global callback space.

Specifies the global space for **globus\_callback\_space\_register\_oneshot()** (p. 7) all other arguments are the same as specified there.

See also:

**globus\_callback\_space\_register\_oneshot()** (p. 7)

#### 4.2.2.7 **#define globus\_callback\_register\_periodic(callback\_handle, delay\_time, period, callback\_func, callback\_user\_arg)**

Register a periodic function in the global callback space.

Specifies the global space for **globus\_callback\_space\_register\_periodic()** (p. 8) all other arguments are the same as specified there.

See also:

**globus\_callback\_space\_register\_periodic()** (p. 8)

#### 4.2.2.8 **#define globus\_callback\_register\_signal\_handler(signum, persist, callback\_func, callback\_user\_arg)**

Register a signal handler in the global callback space.

Specifies the global space for **globus\_callback\_space\_register\_signal\_handler()** (p. 17) all other arguments are the same as specified there.

See also:

**globus\_callback\_space\_register\_signal\_handler()** (p. 17)

### 4.2.3 Typedef Documentation

#### 4.2.3.1 **typedef void(\*) globus\_callback\_func\_t(void \*user\_arg)**

Globus callback prototype.

This is the signature of the function registered with the **globus\_callback\_register\_\*** calls.

If this is a periodic callback, it is guaranteed that the call canNOT be reentered unless **globus\_thread\_blocking\_space\_will\_block()** is called (explicitly, or implicitly via **globus\_cond\_wait()** (p. 42)). Also, if **globus\_callback\_unregister()** (p. 8) is called to cancel this periodic from within this callback, it is guaranteed that the callback will NOT be queued again

If the function will block at all, the user should call **globus\_callback\_get\_timeout()** (p. 11) to see how long this function can safely block or call **globus\_thread\_blocking\_space\_will\_block()**

**Parameters:**

*user\_arg* The user argument registered with this callback

**Returns:**

- void

See also:

**globus\_callback\_space\_register\_one-shot()** (p. 7)

**globus\_callback\_space\_register\_periodic()** (p. 8)

**globus\_thread\_blocking\_space\_will\_block()**

**globus\_callback\_get\_timeout()** (p. 11)

### 4.2.4 Function Documentation

#### 4.2.4.1 **globus\_result\_t globus\_callback\_space\_register\_one-shot (globus\_callback\_handle\_t \* callback\_handle, const globus\_retime\_t \* delay\_time, globus\_callback\_func\_t callback\_func, void \* callback\_user\_arg, globus\_callback\_space\_t space)**



Register a oneshot some delay from now.

This function registers the `callback_func` to start some `delay_time` from now.

**Parameters:**

***callback\_handle*** Storage for a handle. This may be NULL. If it is NOT NULL, you must unregister the callback to reclaim resources.

***delay\_time*** The relative time from now to fire this callback. If NULL, will fire as soon as possible

***callback\_func*** the user func to call

***callback\_user\_arg*** user arg that will be passed to callback

***space*** The space with which to register this callback

**Returns:**

- GLOBUS\_CALLBACK\_ERROR\_INVALID\_ARGUMENT
- GLOBUS\_CALLBACK\_ERROR\_MEMORY\_ALLOC
- GLOBUS\_SUCCESS

**See also:**

`globus_callback_func_t` (p. 7)

`Globus Callback Spaces` (p. 12)

**4.2.4.2** `globus_result_t globus_callback_space_register_periodic (globus_callback_handle_t * callback_handle, const globus_reftime_t * delay_time, const globus_reftime_t * period, globus_callback_func_t callback_func, void * callback_user_arg, globus_callback_space_t space)`

Register a periodic callback.

This function registers a periodic `callback_func` to start some `delay_time` and run every `period` from then.

**Parameters:**

***callback\_handle*** Storage for a handle. This may be NULL. If it is NOT NULL, you must cancel the periodic to reclaim resources.

***delay\_time*** The relative time from now to fire this callback. If NULL, will fire the first callback as soon as possible

***period*** The relative period of this callback

***callback\_func*** the user func to call

***callback\_user\_arg*** user arg that will be passed to callback

***space*** The space with which to register this callback

**Returns:**

- GLOBUS\_CALLBACK\_ERROR\_INVALID\_ARGUMENT
- GLOBUS\_CALLBACK\_ERROR\_MEMORY\_ALLOC
- GLOBUS\_SUCCESS

**See also:**

`globus_callback_unregister()` (p. 8)

`globus_callback_func_t` (p. 7)

`globus_callback_spaces` (p. 12)

#### 4.2.4.3 **globus\_result\_t globus\_callback\_unregister** (**globus\_callback\_handle\_t** *callback\_handle*, **globus\_callback\_func\_t** *unregister\_callback*, **void \****unreg\_arg*, **globus\_bool\_t \****active*)

Unregister a callback.

This function will cancel a callback and free the resources associated with the callback handle. If the callback was able to be canceled immediately (or if it has already run), GLOBUS\_SUCCESS is returned and it is guaranteed that there are no running instances of the callback.

If the callback is currently running (or unstopably about to be run), then the callback is prevented from being requeued, but, the 'official' cancel is deferred until the last running instance of the callback returns. If you need to know when the callback is guaranteed to have been canceled, pass an unregister callback.

If you would like to know if you unregistered a callback before it ran, pass storage for a boolean 'active'. This will be GLOBUS\_TRUE if callback was running. GLOBUS\_FALSE otherwise.

##### **Parameters:**

*callback\_handle* the handle received from a **globus\_callback\_space\_register\_\***(*call*)

*unregister\_callback* the function to call when the callback has been canceled and there are no running instances of it. This will be delivered to the same space used in the register call.

*unreg\_arg* user arg that will be passed to the unregister callback

*active* storage for an indication of whether the callback was running when this call was made

##### **Returns:**

- GLOBUS\_CALLBACK\_ERROR\_INVALID\_CALLBACK\_HANDLE
- GLOBUS\_CALLBACK\_ERROR\_ALREADY\_CANCELED
- GLOBUS\_SUCCESS

##### **See also:**

**globus\_callback\_space\_register\_periodic()** (p. 8)

**globus\_callback\_func\_t** (p. 7)

#### 4.2.4.4 **globus\_result\_t globus\_callback\_adjust\_oneshot** (**globus\_callback\_handle\_t** *callback\_handle*, **const globus\_retime\_t \****new\_delay*)

Adjust the delay of a oneshot callback.

This function allows a user to adjust the delay of a previously registered callback. It is safe to call this within or outside of the callback that is being modified.

Note if the oneshot has already been fired, this function will still return GLOBUS\_SUCCESS, but won't affect anything.

##### **Parameters:**

*callback\_handle* the handle received from a **globus\_callback\_space\_register\_oneshot()** (p. 7) call

*new\_delay* The new delay from now. If NULL, then callback will be fired as soon as possible.

##### **Returns:**

- GLOBUS\_CALLBACK\_ERROR\_INVALID\_CALLBACK\_HANDLE
- GLOBUS\_CALLBACK\_ERROR\_ALREADY\_CANCELED
- GLOBUS\_SUCCESS

##### **See also:**

**globus\_callback\_space\_register\_periodic()** (p. 8)

#### 4.2.4.5 **globus\_result\_t globus\_callback\_adjust\_period (globus\_callback\_handle\_t *callback\_handle*, const globus\_reltime\_t \* *new\_period*)**

Adjust the period of a periodic callback.

This function allows a user to adjust the period of a previously registered callback. It is safe to call this within or outside of the callback that is being modified.

This func also allows a user to effectively 'suspend' a periodic callback until another time by passing a period of NULL. The callback can later be resumed by passing in a new period.

Note that the callback will not be fired sooner than 'new\_period' from now. A 'suspended' callback must still be unregistered to free its resources.

##### **Parameters:**

*callback\_handle* the handle received from a **globus\_callback\_space\_register\_periodic()** (p. 8) call

*new\_period* The new period. If NULL or globus\_i\_reltime\_infinity, then callback will be 'suspended' as soon as the last running instance of it returns.

##### **Returns:**

- GLOBUS\_CALLBACK\_ERROR\_INVALID\_CALLBACK\_HANDLE
- GLOBUS\_CALLBACK\_ERROR\_ALREADY\_CANCELED
- GLOBUS\_SUCCESS

##### **See also:**

**globus\_callback\_space\_register\_periodic()** (p. 8)

#### 4.2.4.6 **void globus\_callback\_space\_poll (const globus\_abstime\_t \* *timestop*, globus\_callback\_space\_t *space*)**

Poll for ready callbacks.

This function is used to poll for registered callbacks.

For non-threaded builds, callbacks are not/can not be delivered unless this is called. Any call to this can cause callbacks registered with the 'global' space to be fired. Whereas callbacks registered with a user's space will only be delivered when this is called with that space.

For threaded builds, this only needs to be called to poll user spaces with behavior == GLOBUS\_CALLBACK\_SPACE\_BEHAVIOR\_SINGLE. The 'global' space and other user spaces are constantly polled in a separate thread. (If it is called in a threaded build for these spaces, it will just yield its thread)

In general, you never need to call this function directly. It is called (when necessary) by **globus\_cond\_wait()** (p. 42). The only case in which a user may wish to call this explicitly is if the application has no aspirations of ever being built threaded.

This function (when not yielding) will block up to timestop or until **globus\_callback\_signal\_poll()** (p. 11) is called by one of the fired callbacks. It will always try and kick out ready callbacks, regardless of the timestop.

##### **Parameters:**

*timestop* The time to block until. If this is NULL or less than the current time, an attempt to fire only ready callbacks is made (no blocking).

*space* The callback space to poll. Note: regardless of what space is passed here, the 'global' space is also always polled.

##### **Returns:**

- void

**See also:**

**Globus Callback Spaces** (p. 12)  
**globus\_condattr\_setspace()** (p. 44)

#### **4.2.4.7 void globus\_callback\_signal\_poll ()**

Signal the poll.

This function signals **globus\_callback\_space\_poll()** (p. 10) that something has changed and it should return to its caller as soon as possible.

In general, you never need to call this function directly. It is called (when necessary) by **globus\_cond\_signal()** (p. 43) or **globus\_cond\_broadcast**. The only case in which a user may wish to call this explicitly is if the application has no aspirations of ever being built threaded.

**Returns:**

- void

**See also:**

**globus\_callback\_space\_poll()** (p. 10)

#### **4.2.4.8 globus\_bool\_t globus\_callback\_get\_timeout (globus\_reftime\_t \* *time\_left*)**

Get the amount of time left in a callback.

This function retrieves the remaining time a callback is allowed to run. If a callback has already timed out, *time\_left* will be set to zero and **GLOBUS\_TRUE** returned. This function is intended to be called within a callback's stack, but is harmless to call anywhere (will return **GLOBUS\_FALSE** and an infinite *time\_left*)

**Parameters:**

*time\_left* storage for the remaining time.

**Returns:**

- **GLOBUS\_FALSE** if time remaining
- **GLOBUS\_TRUE** if already timed out

#### **4.2.4.9 globus\_bool\_t globus\_callback\_has\_time\_expired ()**

See if there is remaining time in a callback.

This function returns **GLOBUS\_TRUE** if the running time of a callback has already expired. This function is intended to be called within a callback's stack, but is harmless to call anywhere (will return **GLOBUS\_FALSE**)

**Returns:**

- **GLOBUS\_FALSE** if time remaining
- **GLOBUS\_TRUE** if already timed out

#### 4.2.4.10 globus\_bool\_t globus\_callback\_was\_restarted ()

See if a callback has been restarted.

If the callback is a oneshot, this merely means the callback called globus\_thread\_blocking\_space\_will\_block (or globus\_cond\_wait() (p. 42) at some point.

For a periodic, it signifies the same and also that the periodic has been requeued. This means that the callback function may be reentered if the period is short enough (on a threaded build)

#### Returns:

- GLOBUS\_FALSE if not restarted
- GLOBUS\_TRUE if restarted

### 4.3 Globus Callback Spaces

Collaboration diagram for Globus Callback Spaces:



#### Defines

- #define GLOBUS\_CALLBACK\_GLOBAL\_SPACE

#### Enumerations

- enum globus\_callback\_space\_behavior\_t {  
    GLOBUS\_CALLBACK\_SPACE\_BEHAVIOR\_SINGLE,  
    GLOBUS\_CALLBACK\_SPACE\_BEHAVIOR\_SERIALIZED,  
    GLOBUS\_CALLBACK\_SPACE\_BEHAVIOR\_THREADED }

#### Functions

- globus\_result\_t globus\_callback\_space\_init (globus\_callback\_space\_t \*space, globus\_callback\_space\_attr\_t attr)
- globus\_result\_t globus\_callback\_space\_reference (globus\_callback\_space\_t space)
- globus\_result\_t globus\_callback\_space\_destroy (globus\_callback\_space\_t space)
- globus\_result\_t globus\_callback\_space\_attr\_init (globus\_callback\_space\_attr\_t \*attr)
- globus\_result\_t globus\_callback\_space\_attr\_destroy (globus\_callback\_space\_attr\_t attr)
- globus\_result\_t globus\_callback\_space\_attr\_set\_behavior (globus\_callback\_space\_attr\_t attr, globus\_callback\_space\_behavior\_t behavior)
- globus\_result\_t globus\_callback\_space\_attr\_get\_behavior (globus\_callback\_space\_attr\_t attr, globus\_callback\_space\_behavior\_t \*behavior)
- globus\_result\_t globus\_callback\_space\_get (globus\_callback\_space\_t \*space)
- int globus\_callback\_space\_get\_depth (globus\_callback\_space\_t space)
- globus\_bool\_t globus\_callback\_space\_is\_single (globus\_callback\_space\_t space)

### 4.3.1 Detailed Description

### 4.3.2 Define Documentation

#### 4.3.2.1 #define GLOBUS\_CALLBACK\_GLOBAL\_SPACE

Global callback space.

The 'global' space handle.

This is the default space handle implied if no spaces are explicitly created.

### 4.3.3 Enumeration Type Documentation

#### 4.3.3.1 enum globus\_callback\_space\_behavior\_t

Callback space behaviors describe how a space behaves.

In a non-threaded build all spaces exhibit a behavior == `_BEHAVIOR_SINGLE`. Setting a specific behavior in this case is ignored.

In a threaded build, `_BEHAVIOR_SINGLE` retains all the rules and behaviors of a non-threaded build while `_BEHAVIOR_THREADED` makes the space act as the global space.

Setting a space's behavior to `_BEHAVIOR_SINGLE` guarantees that the poll protection will always be there and all callbacks are serialized and only kicked out when polled for. In a threaded build, it is still necessary to poll for callbacks in a `_BEHAVIOR_SINGLE` space. (`globus_cond_wait()` (p. 42) will take care of this for you also)

Setting a space's behavior to `_BEHAVIOR_SERIALIZED` guarantees that the poll protection will always be there and all callbacks are serialized. In a threaded build, it is NOT necessary to poll for callbacks in a `_BEHAVIOR_SERIALIZED` space. Callbacks in this space will be delivered as soon as possible, but only one outstanding (and unblocked) callback will be allowed at any time.

Setting a space's behavior to `_BEHAVIOR_THREADED` allows the user to have the poll protection provided by spaces when built non-threaded, yet, be fully threaded when built threaded (where poll protection is not needed)

#### Enumerator:

***GLOBUS\_CALLBACK\_SPACE\_BEHAVIOR\_SINGLE*** The default behavior.

Indicates that you always want poll protection and single threaded behavior (callbacks need to be explicitly polled for)

***GLOBUS\_CALLBACK\_SPACE\_BEHAVIOR\_SERIALIZED*** Indicates that you want poll protection and all callbacks to be serialized (but they do not need to be polled for in a threaded build).

***GLOBUS\_CALLBACK\_SPACE\_BEHAVIOR\_THREADED*** Indicates that you only want poll protection.

### 4.3.4 Function Documentation

#### 4.3.4.1 globus\_result\_t globus\_callback\_space\_init (globus\_callback\_space\_t \* space, globus\_callback\_space\_attr\_t attr)

Initialize a user space.

This creates a user space.

#### Parameters:

***space*** storage for the initialized space handle. This must be destroyed with `globus_callback_space_destroy()` (p. 14)

***attr*** a space attr describing desired behaviors. If `GLOBUS_NULL`, the default behavior of `GLOBUS_CALLBACK_SPACE_BEHAVIOR_SINGLE` is assumed. This attr is copied into the space, so it is acceptable to destroy the attr as soon as it is no longer needed

**Returns:**

- GLOBUS\_CALLBACK\_ERROR\_INVALID\_ARGUMENT on NULL space
- GLOBUS\_CALLBACK\_ERROR\_MEMORY\_ALLOC
- GLOBUS\_SUCCESS

**See also:**

**globus\_condattr\_setspace()** (p. 44)

**4.3.4.2 globus\_result\_t globus\_callback\_space\_reference (globus\_callback\_space\_t space)**

Take a reference to a space.

A library which has been 'given' a space to provide callbacks on would use this to take a reference on the user's space. This prevents mayhem should a user destroy a space before the library is done with it. This reference should be destroyed with **globus\_callback\_space\_destroy()** (p. 14) (think dup())

**Parameters:**

*space* space to reference

**Returns:**

- GLOBUS\_CALLBACK\_ERROR\_INVALID\_SPACE
- GLOBUS\_SUCCESS

**4.3.4.3 globus\_result\_t globus\_callback\_space\_destroy (globus\_callback\_space\_t space)**

Destroy a reference to a user space.

This will destroy a reference to a previously initialized space. Space will not actually be destroyed until all callbacks registered with this space have been run and unregistered (if the user has a handle to that callback) AND all references (from **globus\_callback\_space\_reference()** (p. 13)) have been destroyed.

**Parameters:**

*space* space to destroy, previously initialized by **globus\_callback\_space\_init()** (p. 13) or referenced with **globus\_callback\_space\_reference()** (p. 13)

**Returns:**

- GLOBUS\_CALLBACK\_ERROR\_INVALID\_SPACE
- GLOBUS\_SUCCESS

**See also:**

**globus\_callback\_space\_init()** (p. 13)

**globus\_callback\_space\_reference()** (p. 13)

**4.3.4.4 globus\_result\_t globus\_callback\_space\_attr\_init (globus\_callback\_space\_attr\_t \* attr)**

Initialize a space attr.

Currently, the only attr to set is the behavior. The default behavior associated with this attr is GLOBUS\_CALLBACK\_SPACE\_BEHAVIOR\_SINGLE

**Parameters:**

*attr* storage for the initialized attr. Must be destroyed with **globus\_callback\_space\_attr\_destroy()** (p. 15)

**Returns:**

- GLOBUS\_CALLBACK\_ERROR\_INVALID\_ARGUMENT on NULL attr
- GLOBUS\_CALLBACK\_ERROR\_MEMORY\_ALLOC
- GLOBUS\_SUCCESS

**4.3.4.5 globus\_result\_t globus\_callback\_space\_attr\_destroy (globus\_callback\_space\_attr\_t attr)**

Destroy a space attr.

**Parameters:**

*attr* attr to destroy, previously initialized with **globus\_callback\_space\_attr\_init()** (p. 14)

**Returns:**

- GLOBUS\_CALLBACK\_ERROR\_INVALID\_ARGUMENT on NULL attr
- GLOBUS\_SUCCESS

**See also:**

**globus\_callback\_space\_attr\_init()** (p. 14)

**4.3.4.6 globus\_result\_t globus\_callback\_space\_attr\_set\_behavior (globus\_callback\_space\_attr\_t attr, globus\_callback\_space\_behavior\_t behavior)**

Set the behavior of a space.

**Parameters:**

*attr* attr to associate behavior with  
*behavior* desired behavior

**Returns:**

- GLOBUS\_CALLBACK\_ERROR\_INVALID\_ARGUMENT
- GLOBUS\_SUCCESS

**See also:**

**globus\_callback\_space\_behavior\_t** (p. 12)

**4.3.4.7 globus\_result\_t globus\_callback\_space\_attr\_get\_behavior (globus\_callback\_space\_attr\_t attr, globus\_callback\_space\_behavior\_t \* behavior)**

Get the behavior associated with an attr.

Note: for a non-threaded build, this will always pass back a behavior == GLOBUS\_CALLBACK\_SPACE\_BEHAVIOR\_SINGLE.

**Parameters:**

*attr* attr on which to query behavior



*behavior* storage for the behavior

**Returns:**

- GLOBUS\_CALLBACK\_ERROR\_INVALID\_ARGUMENT
- GLOBUS\_SUCCESS

**4.3.4.8 globus\_result\_t globus\_callback\_space\_get (globus\_callback\_space\_t \* *space*)**

Retrieve the space of a currently running callback.

**Parameters:**

*space* storage for the handle to the space currently running

**Returns:**

- GLOBUS\_CALLBACK\_ERROR\_INVALID\_ARGUMENT on NULL space
- GLOBUS\_CALLBACK\_ERROR\_NO\_ACTIVE\_CALLBACK
- GLOBUS\_SUCCESS

**4.3.4.9 int globus\_callback\_space\_get\_depth (globus\_callback\_space\_t *space*)**

Retrieve the current nesting level of a space.

**Parameters:**

*space* The space to query.

**Returns:**

- the current nesting level
- -1 on invalid space

**4.3.4.10 globus\_bool\_t globus\_callback\_space\_is\_single (globus\_callback\_space\_t *space*)**

See if the specified space is a single threaded behavior space.

**Parameters:**

*space* the space to query

**Returns:**

- GLOBUS\_TRUE if space's behavior is \_BEHAVIOR\_SINGLE
- GLOBUS\_FALSE otherwise

## **4.4 Globus Callback Signal Handling**

**Defines**

- #define GLOBUS\_SIGNAL\_INTERRUPT

## Functions

- `globus_result_t globus_callback_space_register_signal_handler (int signum, globus_bool_t persist, globus_callback_func_t callback_func, void *callback_user_arg, globus_callback_space_t space)`
- `globus_result_t globus_callback_unregister_signal_handler (int signum, globus_callback_func_t unregister_callback, void *unreg_arg)`
- `void globus_callback_add_wakeup_handler (void(*wakeup)(void *), void *user_arg)`

### 4.4.1 Detailed Description

### 4.4.2 Define Documentation

#### 4.4.2.1 #define GLOBUS\_SIGNAL\_INTERRUPT

Use this to trap interrupts (SIGINT on unix). In the future, this will also map to handle ctrl-C on win32.

### 4.4.3 Function Documentation

#### 4.4.3.1 `globus_result_t globus_callback_space_register_signal_handler (int signum, globus_bool_t persist, globus_callback_func_t callback_func, void * callback_user_arg, globus_callback_space_t space)`

Fire a callback when the specified signal is received.

Note that there is a tiny delay between the time this call returns and the signal is actually handled by this library. It is likely that, if the signal was received the instant the call returned, it will be lost (this is normally not an issue, since you would call this in your startup code anyway)

#### Parameters:

*signum* The signal to receive. The following signals are not allowed: SIGKILL, SIGSEGV, SIGABRT, SIGBUS, SIGFPE, SIGILL, SIGIOT, SIGPIPE, SIGEMT, SIGSYS, SIGTRAP, SIGSTOP, SIGCONT, and SIGWAITING

*persist* If GLOBUS\_TRUE, keep this callback registered for multiple signals. If GLOBUS\_FALSE, the signal handler will automatically be unregistered once the signal has been received.

*callback\_func* the user func to call when a signal is received

*callback\_user\_arg* user arg that will be passed to callback

*space* the space to deliver callbacks to.

#### Returns:

- GLOBUS\_CALLBACK\_ERROR\_INVALID\_SPACE
- GLOBUS\_CALLBACK\_ERROR\_INVALID\_ARGUMENT
- GLOBUS\_SUCCESS otherwise

#### 4.4.3.2 `globus_result_t globus_callback_unregister_signal_handler (int signum, globus_callback_func_t unregister_callback, void * unreg_arg)`

Unregister a signal handling callback.

#### Parameters:

*signum* The signal to unregister.

*unregister\_callback* the function to call when the callback has been canceled and there are no running instances of it (may be NULL). This will be delivered to the same space used in the register call.

*unreg\_arg* user arg that will be passed to callback

#### Returns:

- **GLOBUS\_CALLBACK\_ERROR\_INVALID\_ARGUMENT** if this signal was registered with `persist == false`, then there is a race between a signal actually being caught and therefor automatically unregistered and the attempt to manually unregister it. If that race occurs, you will receive this error just as you would for any signal not registered.
- **GLOBUS\_SUCCESS** otherwise

#### 4.4.3.3 void globus\_callback\_add\_wakeup\_handler (void(\*) (void \*) wakeup, void \* user\_arg)

Register a wakeup handler with callback library.

This is really only needed in non-threaded builds, but for cross builds should be used everywhere that a callback may sleep for an extended period of time.

An example use is for an io poller that sleeps indefinitely on `select()`. If the callback library receives a signal that it needs to deliver asap, it will call the wakeup handler(s). These wakeup handlers must run as though they were called from a signal handler (don't use any thread utilities). The io poll example will likely write a single byte to a pipe that `select()` is monitoring.

This handler will not be unregistered until the callback library is deactivated (via `common`).

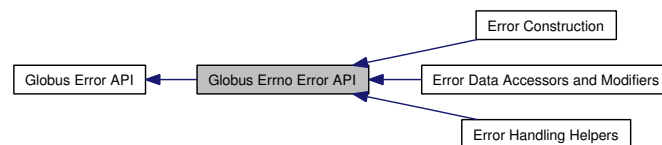
#### Parameters:

*wakeup* function to call when callback library needs you to return asap from any blocked callbacks.

*user\_arg* user data that will be passed along in the wakeup handler

## 4.5 Globus Errno Error API

Collaboration diagram for Globus Errno Error API:



These `globus_error` functions are motivated by the desire to provide a easier way of generating new error types, while at the same time preserving all features (e.g.

#### Modules

- **Error Construction**
- **Error Data Accessors and Modifiers**
- **Error Handling Helpers**

#### 4.5.1 Detailed Description

These `globus_error` functions are motivated by the desire to provide a easier way of generating new error types, while at the same time preserving all features (e.g.

memory management, chaining) of the current error handling framework. The functions in this API are auxiliary to the function in the Globus Generic Error API in the sense that they provide a wrapper for representing system errors in terms of a `globus_error_t`.

Any program that uses Globus Errno Error functions must include "globus\_common.h".

## 4.6 Error Construction

Collaboration diagram for Error Construction:



Create and initialize a Globus Errno Error object.

### Construct Error

- `globus_object_t * globus_error_construct_errno_error (globus_module_descriptor_t *base_source, globus_object_t *base_cause, const int system_errno)`

### Initialize Error

- `globus_object_t * globus_error_initialize_errno_error (globus_object_t *error, globus_module_descriptor_t *base_source, globus_object_t *base_cause, const int system_errno)`

### Defines

- `#define GLOBUS_ERROR_TYPE_ERRNO`

#### 4.6.1 Detailed Description

Create and initialize a Globus Errno Error object.

This section defines operations to create and initialize Globus Errno Error objects.

#### 4.6.2 Define Documentation

##### 4.6.2.1 #define GLOBUS\_ERROR\_TYPE\_ERRNO

Error type definition.

#### 4.6.3 Function Documentation

##### 4.6.3.1 `globus_object_t* globus_error_construct_errno_error (globus_module_descriptor_t * base_source, globus_object_t * base_cause, const int system_errno)`

Allocate and initialize an error of type `GLOBUS_ERROR_TYPE_ERRNO`.

#### Parameters:

*base\_source* Pointer to the originating module.

*base\_cause* The error object causing the error. If this is the original error, this parameter may be NULL.

*system\_errno* The system errno.

#### Returns:

The resulting error object. It is the user's responsibility to eventually free this object using `globus_object_free()`. A `globus_result_t` may be obtained by calling `globus_error_put()` on this object.

#### 4.6.3.2 `globus_object_t* globus_error_initialize_errno_error (globus_object_t * error, globus_module_descriptor_t * base_source, globus_object_t * base_cause, const int system_errno)`

Initialize a previously allocated error of type `GLOBUS_ERROR_TYPE_ERRNO`.

#### Parameters:

*error* The previously allocated error object.

*base\_source* Pointer to the originating module.

*base\_cause* The error object causing the error. If this is the original error this parameter may be `NULL`.

*system\_errno* The system errno.

#### Returns:

The resulting error object. You may have to call `globus_error_put()` on this object before passing it on.

## 4.7 Error Data Accessors and Modifiers

Collaboration diagram for Error Data Accessors and Modifiers:



Get and set data in a Globus Errno Error object.

#### Get Errno

- `int globus_error_errno_get_errno (globus_object_t *error)`

#### Set Errno

- `void globus_error_errno_set_errno (globus_object_t *error, const int system_errno)`

#### 4.7.1 Detailed Description

Get and set data in a Globus Errno Error object.

This section defines operations for accessing and modifying data in a Globus Errno Error object.

#### 4.7.2 Function Documentation

##### 4.7.2.1 `int globus_error_errno_get_errno (globus_object_t * error)`

Retrieve the system errno from a errno error object.

**Parameters:**

*error* The error from which to retrieve the errno

**Returns:**

The errno stored in the object

**4.7.2.2 void globus\_error\_errno\_set\_errno (globus\_object\_t \* *error*, const int *system\_errno*)**

Set the errno in a errno error object.

**Parameters:**

*error* The error object for which to set the errno

*system\_errno* The system errno

**Returns:**

void

**4.8 Error Handling Helpers**

Collaboration diagram for Error Handling Helpers:



Helper functions for dealing with Globus Errno Error objects.

**Error Match**

- globus\_bool\_t **globus\_error\_errno\_match** (globus\_object\_t \**error*, globus\_module\_descriptor\_t \**module*, int *system\_errno*)

**Wrap Errno Error**

- globus\_object\_t \* **globus\_error\_wrap\_errno\_error** (globus\_module\_descriptor\_t \**base\_source*, int *system\_errno*, int *type*, const char \**source\_file*, const char \**source\_func*, int *source\_line*, const char \**short\_desc\_format*,...)

**4.8.1 Detailed Description**

Helper functions for dealing with Globus Errno Error objects.

This section defines utility functions for dealing with Globus Errno Error objects.

**4.8.2 Function Documentation****4.8.2.1 globus\_bool\_t globus\_error\_errno\_match (globus\_object\_t \* *error*, globus\_module\_descriptor\_t \* *module*, int *system\_errno*)**

Check whether the error originated from a specific module and matches a specific errno.

This function checks whether the error or any of it's causative errors originated from a specific module and contains a specific errno. If the module descriptor is left unspecified this function will check for any error of the specified errno and vice versa.

#### Parameters:

***error*** The error object for which to perform the check  
***module*** The module descriptor to check for  
***system\_errno*** The errno to check for

#### Returns:

GLOBUS\_TRUE - the error matched the module and errno GLOBUS\_FALSE - the error failed to match the module and errno

#### 4.8.2.2 globus\_object\_t\* globus\_error\_wrap\_errno\_error (globus\_module\_descriptor\_t \*base\_source, int system\_errno, int type, const char \*source\_file, const char \*source\_func, int source\_line, const char \*short\_desc\_format, ...)

Allocate and initialize an error of type GLOBUS\_ERROR\_TYPE\_GLOBUS which contains a causal error of type GLOBUS\_ERROR\_TYPE\_ERRNO.

#### Parameters:

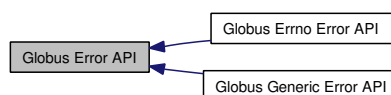
***base\_source*** Pointer to the originating module.  
***system\_errno*** The errno to use when generating the causal error.  
***type*** The error type. We may reserve part of this namespace for common errors. Errors not in this space are assumed to be local to the originating module.  
***source\_file*** Name of file. Use \_\_FILE\_\_  
***source\_func*** Name of function. Use \_globus\_func\_name and declare your func with GlobusFunc-Name(<name>)  
***source\_line*** Line number. Use \_\_LINE\_\_  
***short\_desc\_format*** Short format string giving a succinct description of the error. To be passed on to the user.  
**...** Arguments for the format string.

#### Returns:

The resulting error object. It is the user's responsibility to eventually free this object using globus\_object\_free(). A globus\_result\_t may be obtained by calling globus\_error\_put() on this object.

## 4.9 Globus Error API

Collaboration diagram for Globus Error API:



Intended use:.

## Modules

- **Globus Errno Error API**
- **Globus Generic Error API**

### 4.9.1 Detailed Description

Intended use:.

If a function needs to return an error it should do the following:

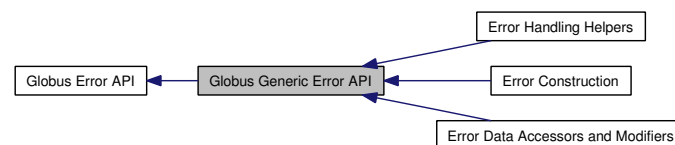
- External errors, such as error returns from system calls and GSSAPI errors, should be wrapped using the appropriate error type.
- The wrapped external error should then be passed as the cause of a globus error.
- External error types are expected to provide a utility function to combine the above two steps.
- The globus error should then be returned from the function.

Notes on how to generate globus errors:

- Module specific error types should be greater or equal to 1024 (to leave some space for global error types).
- You may wish to generate a mapping from error types to format strings for use in short descriptions.
- You may also wish to generate a common prefix for all of the above format strings. The suggested prefix is "Function: s Line: s ".

## 4.10 Globus Generic Error API

Collaboration diagram for Globus Generic Error API:



These globus\_error functions are motivated by the desire to provide a easier way of generating new error types, while at the same time preserving all features (e.g.

## Modules

- **Error Construction**
- **Error Data Accessors and Modifiers**
- **Error Handling Helpers**



### 4.10.1 Detailed Description

These globus\_error functions are motivated by the desire to provide a easier way of generating new error types, while at the same time preserving all features (e.g.

memory management, chaining) of the current error handling framework. It does this by defining a generic error type for globus which in turn contains a integer in it's instance data which is used for carrying the actual error type information.

Any program that uses Globus Generic Error functions must include "globus\_common.h".

## 4.11 Error Construction

Collaboration diagram for Error Construction:



Create and initialize a Globus Generic Error object.

### Construct Error

- `globus_object_t * globus_error_construct_error (globus_module_descriptor_t *base_source, globus_object_t *base_cause, int type, const char *source_file, const char *source_func, int source_line, const char *short_desc_format,...)`
- `globus_object_t * globus_error_v_construct_error (globus_module_descriptor_t *base_source, globus_object_t *base_cause, const int type, const char *source_file, const char *source_func, int source_line, const char *short_desc_format, va_list ap)`

### Initialize Error

- `globus_object_t * globus_error_initialize_error (globus_object_t *error, globus_module_descriptor_t *base_source, globus_object_t *base_cause, int type, const char *source_file, const char *source_func, int source_line, const char *short_desc_format, va_list ap)`

### Defines

- `#define GLOBUS_ERROR_TYPE_GLOBUS`

### 4.11.1 Detailed Description

Create and initialize a Globus Generic Error object.

This section defines operations to create and initialize Globus Generic Error objects.

### 4.11.2 Define Documentation

#### 4.11.2.1 #define GLOBUS\_ERROR\_TYPE\_GLOBUS

Error type definition.

### 4.11.3 Function Documentation

**4.11.3.1 globus\_object\_t\* globus\_error\_construct\_error (globus\_module\_descriptor\_t \* *base\_source*, globus\_object\_t \* *base\_cause*, int *type*, const char \* *source\_file*, const char \* *source\_func*, int *source\_line*, const char \* *short\_desc\_format*, ...)**

Allocate and initialize an error of type GLOBUS\_ERROR\_TYPE\_GLOBUS.

#### Parameters:

*base\_source* Pointer to the originating module.

*base\_cause* The error object causing the error. If this is the original error this parameter may be NULL.

*type* The error type. We may reserve part of this namespace for common errors. Errors not in this space are assumed to be local to the originating module.

*source\_file* Name of file. Use `__FILE__`

*source\_func* Name of function. Use `_globus_func_name` and declare your func with `GlobusFunc-Name(<name>)`

*source\_line* Line number. Use `__LINE__`

*short\_desc\_format* Short format string giving a succinct description of the error. To be passed on to the user.

... Arguments for the format string.

#### Returns:

The resulting error object. It is the user's responsibility to eventually free this object using `globus_object_free()`. A `globus_result_t` may be obtained by calling `globus_error_put()` on this object.

**4.11.3.2 globus\_object\_t\* globus\_error\_v\_construct\_error (globus\_module\_descriptor\_t \* *base\_source*, globus\_object\_t \* *base\_cause*, const int *type*, const char \* *source\_file*, const char \* *source\_func*, int *source\_line*, const char \* *short\_desc\_format*, va\_list *ap*)**

Allocate and initialize an error of type GLOBUS\_ERROR\_TYPE\_GLOBUS.

#### Parameters:

*base\_source* Pointer to the originating module.

*base\_cause* The error object causing the error. If this is the original error this parameter may be NULL.

*type* The error type. We may reserve part of this namespace for common errors. Errors not in this space are assumed to be local to the originating module.

*source\_file* Name of file. Use `__FILE__`

*source\_func* Name of function. Use `_globus_func_name` and declare your func with `GlobusFunc-Name(<name>)`

*source\_line* Line number. Use `__LINE__`

*short\_desc\_format* Short format string giving a succinct description of the error. To be passed on to the user.

*ap* Arguments for the format string.

#### Returns:

The resulting error object. It is the user's responsibility to eventually free this object using `globus_object_free()`. A `globus_result_t` may be obtained by calling `globus_error_put()` on this object.

**4.11.3.3 globus\_object\_t\* globus\_error\_initialize\_error (globus\_object\_t \* *error*, globus\_module\_descriptor\_t \* *base\_source*, globus\_object\_t \* *base\_cause*, int *type*, const char \* *source\_file*, const char \* *source\_func*, int *source\_line*, const char \* *short\_desc\_format*, va\_list *ap*)**

Initialize a previously allocated error of type GLOBUS\_ERROR\_TYPE\_GLOBUS.

#### Parameters:

***error*** The previously allocated error object.

***base\_source*** Pointer to the originating module.

***base\_cause*** The error object causing the error. If this is the original error this parameter may be NULL.

***type*** The error type. We may reserve part of this namespace for common errors. Errors not in this space are assumed to be local to the originating module.

***source\_file*** Name of file. Use `__FILE__`

***source\_func*** Name of function. Use `_globus_func_name` and declare your func with `GlobusFuncName(<name>)`

***source\_line*** Line number. Use `__LINE__`

***short\_desc\_format*** Short format string giving a succinct description of the error. To be passed on to the user.

***ap*** Arguments for the format string.

#### Returns:

The resulting error object. You may have to call `globus_error_put()` on this object before passing it on.

## 4.12 Error Data Accessors and Modifiers

Collaboration diagram for Error Data Accessors and Modifiers:



Get and set data in a Globus Generic Error object.

#### Get Source

- `globus_module_descriptor_t * globus_error_get_source (globus_object_t *error)`

#### Set Source

- `void globus_error_set_source (globus_object_t *error, globus_module_descriptor_t *source_module)`

#### Get Cause

- `globus_object_t * globus_error_get_cause (globus_object_t *error)`

#### Set Cause

- `void globus_error_set_cause (globus_object_t *error, globus_object_t *causal_error)`

### Get Type

- `int globus_error_get_type (globus_object_t *error)`

### Set Type

- `void globus_error_set_type (globus_object_t *error, const int type)`

### Get Short Description

- `char * globus_error_get_short_desc (globus_object_t *error)`

### Set Short Description

- `void globus_error_set_short_desc (globus_object_t *error, const char *short_desc_format,...)`

### Get Long Description

- `char * globus_error_get_long_desc (globus_object_t *error)`

### Set Long Description

- `void globus_error_set_long_desc (globus_object_t *error, const char *long_desc_format,...)`

## 4.12.1 Detailed Description

Get and set data in a Globus Generic Error object.

This section defines operations for accessing and modifying data in a Globus Generic Error object.

## 4.12.2 Function Documentation

### 4.12.2.1 `globus_module_descriptor_t* globus_error_get_source (globus_object_t * error)`

Retrieve the originating module descriptor from a error object.

#### Parameters:

***error*** The error from which to retrieve the module descriptor

#### Returns:

The originating module descriptor.

### 4.12.2.2 `void globus_error_set_source (globus_object_t * error, globus_module_descriptor_t * source_module)`

Set the originating module descriptor in a error object.

#### Parameters:

***error*** The error object for which to set the causative error

*source\_module* The originating module descriptor

**Returns:**

void

**4.12.2.3 globus\_object\_t\* globus\_error\_get\_cause (globus\_object\_t \* *error*)**

Retrieve the underlying error from a error object.

**Parameters:**

*error* The error from which to retrieve the causative error.

**Returns:**

The underlying error object if it exists, NULL if it doesn't.

**4.12.2.4 void globus\_error\_set\_cause (globus\_object\_t \* *error*, globus\_object\_t \* *causal\_error*)**

Set the causative error in a error object.

**Parameters:**

*error* The error object for which to set the causative error.

*causal\_error* The causative error.

**Returns:**

void

**4.12.2.5 int globus\_error\_get\_type (globus\_object\_t \* *error*)**

Retrieve the error type from a generic globus error object.

**Parameters:**

*error* The error from which to retrieve the error type

**Returns:**

The error type of the object

**4.12.2.6 void globus\_error\_set\_type (globus\_object\_t \* *error*, const int *type*)**

Set the error type in a generic globus error object.

**Parameters:**

*error* The error object for which to set the error type

*type* The error type

**Returns:**

void

#### 4.12.2.7 `char* globus_error_get_short_desc (globus_object_t * error)`

Retrieve the short error description from a generic globus error object.

##### Parameters:

*error* The error from which to retrieve the description

##### Returns:

The short error description of the object

#### 4.12.2.8 `void globus_error_set_short_desc (globus_object_t * error, const char * short_desc_format, ...)`

Set the short error description in a generic globus error object.

##### Parameters:

*error* The error object for which to set the description

*short\_desc\_format* Short format string giving a succinct description of the error. To be passed on to the user.

... Arguments for the format string.

##### Returns:

void

#### 4.12.2.9 `char* globus_error_get_long_desc (globus_object_t * error)`

Retrieve the long error description from a generic globus error object.

##### Parameters:

*error* The error from which to retrieve the description

##### Returns:

The long error description of the object

#### 4.12.2.10 `void globus_error_set_long_desc (globus_object_t * error, const char * long_desc_format, ...)`

Set the long error description in a generic globus error object.

##### Parameters:

*error* The error object for which to set the description

*long\_desc\_format* Longer format string giving a more detailed explanation of the error.

##### Returns:

void

## 4.13 Error Handling Helpers

Collaboration diagram for Error Handling Helpers:



Helper functions for dealing with Globus Generic Error objects.

## Error Match

- `globus_bool_t globus_error_match (globus_object_t *error, globus_module_descriptor_t *module, int type)`

## Print Error Chain

- `char * globus_error_print_chain (globus_object_t *error)`

## Print User Friendly Error Message

- `char * globus_error_print_friendly (globus_object_t *error)`

### 4.13.1 Detailed Description

Helper functions for dealing with Globus Generic Error objects.

This section defines utility functions for dealing with Globus Generic Error objects.

### 4.13.2 Function Documentation

#### 4.13.2.1 `globus_bool_t globus_error_match (globus_object_t * error, globus_module_descriptor_t * module, int type)`

Check whether the error originated from a specific module and is of a specific type.

This function checks whether the error or any of its causative errors originated from a specific module and is of a specific type. If the module descriptor is left unspecified this function will check for any error of the specified type and vice versa.

#### Parameters:

- error* The error object for which to perform the check
- module* The module descriptor to check for
- type* The type to check for

#### Returns:

GLOBUS\_TRUE - the error matched the module and type GLOBUS\_FALSE - the error failed to match the module and type

#### 4.13.2.2 `char* globus_error_print_chain (globus_object_t * error)`

Return a string containing all printable errors found in a error object and its causative error chain.

If the GLOBUS\_ERROR\_VERBOSE env is set, file, line and function info will also be printed (where available). Otherwise, only the module name will be printed.

#### Parameters:

- error* The error to print

#### Returns:

A string containing all printable errors. This string needs to be freed by the user of this function.

#### 4.13.2.3 char\* globus\_error\_print\_friendly (globus\_object\_t \* error)

Return a string containing error messages from the top 1 and bottom 3 objects, and, if found, show a friendly error message.

The error chain will be searched from top to bottom until a friendly handler is found and a friendly message is created.

If the GLOBUS\_ERROR\_VERBOSE env is set, then the result from **globus\_error\_print\_chain()** (p. 30) will be used.

##### Parameters:

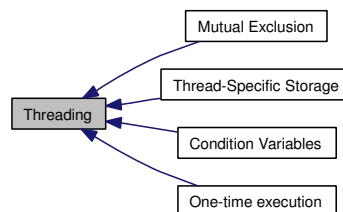
**error** The error to print

##### Returns:

A string containing a friendly error message. This string needs to be freed by the user of this function.

## 4.14 Threading

Collaboration diagram for Threading:



The Globus runtime includes support for portably creating threads on POSIX and Windows systems.

##### Modules

- **Mutual Exclusion**
- **Condition Variables**
- **Thread-Specific Storage**
- **One-time execution**

##### Data Structures

- **union globus\_thread\_t**  
*Thread ID.*
- **union globus\_threadattr\_t**  
*Thread attributes.*
- **union globus\_thread\_key\_t**  
*Thread-specific data key.*



## Defines

- `#define GLOBUS_THREAD_CANCEL_DISABLE 0`
- `#define GLOBUS_THREAD_CANCEL_ENABLE 1`
- `#define GLOBUS_THREAD_MODULE (&globus_i_thread_module)`

## Typedefs

- `typedef void(*) globus_thread_key_destructor_func_t (void *value)`

## Functions

- `int globus_thread_set_model (const char *model)`
- `int globus_thread_create (globus_thread_t *thread, globus_threadattr_t *attr, globus_thread_func_t func, void *user_arg)`
- `void globus_thread_yield (void)`
- `void globus_thread_exit (void *value)`
- `int globus_thread_sigmask (int how, const sigset_t *new_mask, sigset_t *old_mask)`
- `int globus_thread_kill (globus_thread_t thread, int sig)`
- `globus_thread_t globus_thread_self (void)`
- `globus_bool_t globus_thread_equal (globus_thread_t thread1, globus_thread_t thread2)`
- `globus_bool_t globus_thread_preemptive_threads (void)`
- `globus_bool_t globus_i_am_only_thread (void)`
- `void * globus_thread_cancellable_func (void *(*func)(void *), void *arg, void(*cleanup_func)(void *), void *cleanup_arg, globus_bool_t execute_cleanup)`
- `int globus_thread_cancel (globus_thread_t thr)`
- `void globus_thread_testcancel (void)`
- `int globus_thread_setcancelstate (int state, int *oldstate)`

### 4.14.1 Detailed Description

The Globus runtime includes support for portably creating threads on POSIX and Windows systems.

It also provides a callback-driven system for applications that may use threads but don't require them. The Globus Thread API is modeled closely after the POSIX threads API.

Applications can choose whether to run as threaded or non-threaded at runtime by either setting the `GLOBUS_THREAD_MODEL` environment variable or calling the `globus_thread_set_model()` (p. 33) function prior to activating any Globus modules.

The Globus thread system provides primitives for mutual exclusion (`globus_mutex_t` (p. 52), `globus_rmutex_t` (p. 53), `globus_rw_mutex_t`), event synchronization (`globus_cond_t` (p. 52)), one-time execution (`globus_once_t`), and threading (`globus_thread_t` (p. 53)).

In non-threaded operation, `globus_cond_wait()` (p. 42) and its variants will poll the callback queue and I/O system to allow event-driven programs to run in the absence of threads. The `globus_thread_create()` (p. 33) function will fail in that model. Other primitive operations will return success but not provide any thread exclusion as there is only one thread.

### 4.14.2 Define Documentation

#### 4.14.2.1 `#define GLOBUS_THREAD_CANCEL_DISABLE 0`

Disable thread cancellation value.

See also:

`globus_thread_setcancelstate()` (p. 36)

#### 4.14.2.2 `#define GLOBUS_THREAD_CANCEL_ENABLE 1`

Enable thread cancellation value.

See also:

`globus_thread_setcancelstate()` (p. 36)

#### 4.14.2.3 `#define GLOBUS_THREAD_MODULE (&globus_i_thread_module)`

Thread Module.

### 4.14.3 Typedef Documentation

#### 4.14.3.1 `typedef void(*) globus_thread_key_destructor_func_t(void *value)`

Thread-specific data destructor.

### 4.14.4 Function Documentation

#### 4.14.4.1 `int globus_thread_set_model (const char * model)`

Select threading model for an application.

The `globus_thread_set_model()` (p. 33) function selects which runtime model the current application will use. By default, the Globus runtime uses a non-threaded model. Additional models may be available based on system support: pthread, or windows. This function must be called prior to activating any globus module, as it changes how certain functions (like `globus_mutex_lock()` (p. 38) and `globus_cond_wait()` (p. 42)) behave. This function overrides the value set by the GLOBUS\_THREAD\_MODEL environment variable.

The `globus_thread_set_model()` (p. 33) function will fail if a Globus module has been activated already.

#### Parameters:

*model* The name of the thread model to use. Depending on operating system capabilities, this may be "none", "pthread", "windows", or some other custom thread implementation. The corresponding libtool module "libglobus\_thread\_pthread.la" or "libglobus\_thread\_windows.la" must be installed on the system for it to be used.

#### Returns:

On success, `globus_thread_set_model()` (p. 33) sets the name of the thread model to use and returns GLOBUS\_SUCCESS. If an error occurs, then `globus_thread_set_model()` (p. 33) returns GLOBUS\_FAILURE.

#### 4.14.4.2 `int globus_thread_create (globus_thread_t * thread, globus_threadattr_t * attr, globus_thread_func_t func, void * user_arg)`

Create a new thread.

The `globus_thread_create()` (p. 33) function creates a new thread of execution in the current process to run the function pointed to by the *func* parameter passed the *user\_arg* value as its only parameter. This new thread will

be detached, so that storage associated with the thread will be automatically reclaimed by the operating system. A thread identifier will be copied to the value pointed to by the *thread* parameter if it is non-NULL. The caller may use this thread identifier to signal or cancel this thread. The *attr* parameter is ignored by this function. If the "none" threading model is used by an application, then this function will always fail. One alternative that will work both with and without threads is to use the functions in the **Globus Callback API** (p. 2).

**Parameters:**

*thread* Pointer to a variable to contain the new thread's identifier.

*attr* Ignored

*func* Pointer to a function to start in the new thread.

*user\_arg* Argument to the new thread's function.

**Returns:**

On success, **globus\_thread\_create()** (p. 33) will start a new thread, invoking (\*func)(user\_arg), modify the value pointed to by the *thread* parameter to contain the new thread's identifier and return GLOBUS\_SUCCESS. If an error occurs, then the value of *thread* is undefined and **globus\_thread\_create()** (p. 33) returns an implementation-specific non-zero error value.

#### 4.14.4.3 void globus\_thread\_yield (void)

Yield execution to another thread.

The **globus\_thread\_yield()** (p. 34) function yields execution to other threads which are ready for execution. The current thread may continue to execute if there are no other threads in the system's ready queue.

#### 4.14.4.4 void globus\_thread\_exit (void \* value)

Terminate the current thread

The **globus\_thread\_exit()** (p. 34) terminates the current thread with the value passed to it.

This function does not return.

#### 4.14.4.5 int globus\_thread\_sigmask (int how, const sigset\_t \* new\_mask, sigset\_t \* old\_mask)

Modify the current thread's signal mask.

The **globus\_thread\_sigmask()** (p. 34) function modifies the current thread's signal mask and returns the old value of the signal mask in the value pointed to by the *old\_mask* parameter. The *how* parameter can be one of SIG\_BLOCK, SIG\_UNBLOCK, or SIG\_SETMASK to control how the signal mask is modified.

**Parameters:**

*how* Flag indicating how to interpret *new\_mask* if it is non-NULL. If *how* is SIG\_BLOCK, then all signals in *new\_mask* are blocked, as well as any which were previously blocked. If *how* is SIG\_UNBLOCK, then all signals in which were previously blocked in *new\_mask* are unblocked. If *how* is SIG\_SETMASK, then the old signal mask is replaced with the value of *new\_mask*.

*new\_mask* Set of signals to block or unblock, based on the *how* parameter.

*old\_mask* A pointer to be set to the old signal mask associated with the current thread.

**Returns:**

On success, **globus\_thread\_sigmask()** (p. 34) modifies the signal mask, modifies the value pointed to by *old\_mask* with the signal mask prior to this function's execution and returns GLOBUS\_SUCCESS. If an error occurs, **globus\_thread\_sigmask()** (p. 34) returns an implementation-specific non-zero error value.

#### 4.14.4.6 **int globus\_thread\_kill (globus\_thread\_t thread, int sig)**

Send a signal to a thread.

The **globus\_thread\_kill()** (p. 34) function sends the signal specified by the *sig* number to the thread whose ID matches the *thread* parameter. Depending on the signal mask of that thread, this may result in a signal being delivered or not, and depending on the process's signal actions, a signal handler, termination, or no operation will occur in that thread.

##### **Parameters:**

*thread* The thread identifier of the thread to signal.

*sig* The signal to send to the thread.

##### **Returns:**

On success, **globus\_thread\_kill()** (p. 34) queues the signal for delivery to the specified thread and returns GLOBUS\_SUCCESS. If an error occurs, **globus\_thread\_kill()** (p. 34) returns an implementation-specific non-zero error value.

#### 4.14.4.7 **globus\_thread\_t globus\_thread\_self (void)**

Determine the current thread's ID.

The **globus\_thread\_self()** (p. 35) function returns the thread identifier of the current thread. This value is unique among all threads which are running at any given time.

#### 4.14.4.8 **globus\_bool\_t globus\_thread\_equal (globus\_thread\_t thread1, globus\_thread\_t thread2)**

Check whether thread identifiers match.

The **globus\_thread\_equal()** (p. 35) function checks whether the thread identifiers passed as the *thread1* and *thread2* parameters refer to the same thread. If so, **globus\_thread\_equal()** (p. 35) returns GLOBUS\_TRUE; otherwise GLOBUS\_FALSE.

##### **Parameters:**

*thread1* Thread identifier to compare.

*thread2* Thread identifier to compare.

#### 4.14.4.9 **globus\_bool\_t globus\_thread\_preemptive\_threads (void)**

Indicate whether the active thread model supports preemption.

The **globus\_thread\_preemptive\_threads()** (p. 35) function returns GLOBUS\_TRUE if the current thread model supports thread preemption; otherwise it returns GLOBUS\_FALSE.

#### 4.14.4.10 **globus\_bool\_t globus\_i\_am\_only\_thread (void)**

Determine if threads are supported.

The **globus\_i\_am\_only\_thread()** (p. 35) function returns GLOBUS\_TRUE if the current thread model is the "none" thread model; GLOBUS\_FALSE otherwise. If running with the "none" thread model, there will only be one Globus thread available and the **globus\_thread\_create()** (p. 33) function will always fail.

#### 4.14.4.11 void\* globus\_thread\_cancellable\_func (void (\*)(void \*) *func*, void \* *arg*, void (\*)(void \*) *cleanup\_func*, void \* *cleanup\_arg*, globus\_bool\_t *execute\_cleanup*)

Execute a function with thread cleanup in case of cancellation.

The **globus\_thread\_cancellable\_func()** (p. 35) function provides an interface to POSIX thread cancellation points that does not rely on preprocessor macros. It is roughly equivalent to

```
pthread_cleanup_push(cleanup_func, cleanup_arg);
(*func)(arg);
pthread_cleanup_pop(execute_cleanup)
```

##### Parameters:

***func*** Pointer to a function which may be cancelled.

***arg*** Parameter to the *func* function.

***cleanup\_func*** Pointer to a function to execute if thread cancellation occurs during *func*.

***cleanup\_arg*** Parameter to the *cleanup\_func* function.

***execute\_cleanup*** Flag indicating whether the function pointed to by *cleanup\_func* should be executed after *func* completes even if it is not cancelled.

##### Returns:

**globus\_thread\_cancellable\_func()** (p. 35) returns the value returned by *func*.

#### 4.14.4.12 int globus\_thread\_cancel (globus\_thread\_t *thr*)

Cancel a thread.

The **globus\_thread\_cancel()** (p. 36) function cancels the thread with the identifier *thr* if it is still executing. If it is running with a cancellation cleanup stack, the functions in that stack are executed. The target thread's cancel state determines when the cancellation is delivered.

##### Parameters:

***thr*** The id of the thread to cancel

##### Returns:

On success, the **globus\_thread\_cancel()** (p. 36) function delivers the cancellation to the target thread and returns GLOBUS\_SUCCESS. If an error occurs, **globus\_thread\_cancel()** (p. 36) returns an implementation-specific non-zero error value.

#### 4.14.4.13 void globus\_thread\_testcancel (void)

Thread cancellation point.

The **globus\_thread\_testcancel()** (p. 36) function acts as a cancellation point for the current thread. If a thread has called **globus\_thread\_cancel()** (p. 36) and cancellation is enabled, this will cause the thread to be cancelled and any functions on the thread's cleanup stack to be executed. This function will not return if the thread is cancelled.

#### 4.14.4.14 int globus\_thread\_setcancelstate (int *state*, int \* *oldstate*)

Set the thread's cancellable state.

The **globus\_thread\_setcancelstate()** (p. 36) function sets the current cancellation state to either GLOBUS\_THREAD\_CANCEL\_DISABLE or GLOBUS\_THREAD\_CANCEL\_ENABLE, do control whether **globus\_thread\_cancel()** (p. 36) is able to cancel this thread.

### Parameters:

**state** The desired cancellation state. If the value is `GLOBUS_THREAD_CANCEL_DISABLE`, then cancellation will be disabled for this thread. If the value is `GLOBUS_THREAD_CANCEL_ENABLE`, then cancellation will be enabled for this thread.

**oldstate** A pointer to a value which will be set to the value of the thread's cancellation state when this function call began. This may be `NULL` if the caller is not interested in the previous value.

### Returns:

On success, the `globus_thread_setcancelstate()` (p. 36) function modifies the thread cancellation state, modifies `oldstate` (if non-`NULL`) to the value of its previous state, and returns `GLOBUS_SUCCESS`. If an error occurs, `globus_thread_setcancelstate()` (p. 36) returns an implementation-specific non-zero error value.

## 4.15 Mutual Exclusion

Collaboration diagram for Mutual Exclusion:



### Data Structures

- union `globus_mutex_t`  
*Mutex.*
- union `globus_mutexattr_t`  
*Mutex attribute.*
- struct `globus_rmutex_t`  
*Recursive Mutex.*

### Recursive Mutex

- int `globus_rmutex_init (globus_rmutex_t *rmutex, globus_rmutexattr_t *rattr)`
- int `globus_rmutex_lock (globus_rmutex_t *rmutex)`
- int `globus_rmutex_unlock (globus_rmutex_t *rmutex)`
- int `globus_rmutex_destroy (globus_rmutex_t *rmutex)`

### Typedefs

- typedef int `globus_rmutexattr_t`

### Functions

- int `globus_mutex_init (globus_mutex_t *mutex, globus_mutexattr_t *attr)`
- int `globus_mutex_destroy (globus_mutex_t *mutex)`
- int `globus_mutex_lock (globus_mutex_t *mutex)`
- int `globus_mutex_unlock (globus_mutex_t *mutex)`
- int `globus_mutex_trylock (globus_mutex_t *mutex)`
- int `globus_mutexattr_init (globus_mutexattr_t *attr)`
- int `globus_mutexattr_destroy (globus_mutexattr_t *attr)`

### 4.15.1 Detailed Description

The Globus runtime includes three portable, related mutual exclusion primitives that can be used in applications and libraries. These are

- **globus\_mutex\_t** (p. 52): a non-recursive, non-shared lock
- **globus\_rmutex\_t** (p. 53): a recursive non-shared lock
- **globus\_rw\_mutex\_t**: a reader-writer lock

### 4.15.2 Typedef Documentation

#### 4.15.2.1 typedef int globus\_rmutexattr\_t

Recursive mutex attribute.

### 4.15.3 Function Documentation

#### 4.15.3.1 int globus\_mutex\_init (globus\_mutex\_t \* *mutex*, globus\_mutexattr\_t \* *attr*)

Initialize a mutex.

The **globus\_mutex\_init()** (p. 38) function creates a mutex variable that can be used for synchronization. Currently, the *attr* parameter is ignored.

#### Parameters:

*mutex* Pointer to the mutex to initialize.

*attr* Ignored.

#### Returns:

On success, **globus\_mutex\_init()** (p. 38) initializes the mutex and returns GLOBUS\_SUCCESS. Otherwise, a non-0 value is returned.

#### 4.15.3.2 int globus\_mutex\_destroy (globus\_mutex\_t \* *mutex*)

Destroy a mutex.

The **globus\_mutex\_destroy()** (p. 38) function destroys the mutex pointed to by its *mutex* parameter. After a mutex is destroyed it may no longer be used unless it is again initialized by **globus\_mutex\_init()** (p. 38). Behavior is undefined if **globus\_mutex\_destroy()** (p. 38) is called with a pointer to a locked mutex.

#### Parameters:

*mutex* The mutex to destroy

#### Returns:

On success, **globus\_mutex\_destroy()** (p. 38) returns GLOBUS\_SUCCESS. Otherwise, a non-zero implementation-specific error value is returned.

#### 4.15.3.3 `int globus_mutex_lock (globus_mutex_t * mutex)`

Lock a mutex.

The `globus_mutex_lock()` (p. 38) function locks the mutex pointed to by its *mutex* parameter. Upon successful return, the thread calling `globus_mutex_lock()` (p. 38) has an exclusive lock on the resources protected by *mutex*. Other threads calling `globus_mutex_lock()` (p. 38) will wait until that thread later calls `globus_mutex_unlock()` (p. 39) or `globus_cond_wait()` (p. 42) with that mutex. Depending on the thread model, calling `globus_mutex_lock` on a mutex locked by the current thread will either return an error or result in deadlock.

##### Parameters:

*mutex* The mutex to lock.

##### Returns:

On success, `globus_mutex_lock()` (p. 38) returns `GLOBUS_SUCCESS`. Otherwise, a non-zero implementation-specific error value is returned.

#### 4.15.3.4 `int globus_mutex_unlock (globus_mutex_t * mutex)`

Unlock a mutex.

The `globus_mutex_unlock()` (p. 39) function unlocks the mutex pointed to by its *mutex* parameter. Upon successful return, the thread calling `globus_mutex_unlock()` (p. 39) no longer has an exclusive lock on the resources protected by *mutex*. Another thread calling `globus_mutex_lock()` (p. 38) may be unblocked so that it may acquire the mutex. Behavior is undefined if `globus_mutex_unlock` is called with an unlocked mutex.

##### Parameters:

*mutex* The mutex to unlock.

##### Returns:

On success, `globus_mutex_unlock()` (p. 39) returns `GLOBUS_SUCCESS`. Otherwise, a non-zero implementation-specific error value is returned.

#### 4.15.3.5 `int globus_mutex_trylock (globus_mutex_t * mutex)`

Lock a mutex if it is not locked.

The `globus_mutex_trylock()` (p. 39) function locks the mutex pointed to by its *mutex* parameter if no thread has already locked the mutex. If *mutex* is locked, then `globus_mutex_trylock()` (p. 39) returns `EBUSY` and does not block the current thread or lock the mutex. Upon successful return, the thread calling `globus_mutex_trylock()` (p. 39) has an exclusive lock on the resources protected by *mutex*. Other threads calling `globus_mutex_lock()` (p. 38) will wait until that thread later calls `globus_mutex_unlock()` (p. 39) or `globus_cond_wait()` (p. 42) with that mutex.

##### Parameters:

*mutex* The mutex to lock.

##### Returns:

On success, `globus_mutex_trylock()` (p. 39) returns `GLOBUS_SUCCESS` and locks the mutex. If another thread holds the lock, `globus_mutex_trylock()` (p. 39) returns `EBUSY`. Otherwise, a non-zero implementation-specific error value is returned.



#### 4.15.3.6 `int globus_mutexattr_init (globus_mutexattr_t * attr)`

Initialize a mutex attribute.

The **`globus_mutexattr_init()`** (p. 39) function initializes the mutex attribute structure pointed to by its *attr* parameter. Currently there are no attribute values that can be set via this API, so there's no real use to calling this function.

##### Parameters:

*attr* Attribute structure to initialize.

##### Returns:

Upon success, **`globus_mutexattr_init()`** (p. 39) returns `GLOBUS_SUCCESS` and modifies the attribute pointed to by *attr*. If an error occurs, **`globus_mutexattr_init()`** (p. 39) returns an implementation-specific non-zero error code.

#### 4.15.3.7 `int globus_mutexattr_destroy (globus_mutexattr_t * attr)`

Destroy a mutex attribute.

The **`globus_mutexattr_destroy()`** (p. 40) function destroys the mutex attribute structure pointed to by its *attr* parameter.

##### Parameters:

*attr* Attribute structure to destroy.

##### Returns:

Upon success, **`globus_mutexattr_destroy()`** (p. 40) returns `GLOBUS_SUCCESS` and modifies the attribute pointed to by *attr*. If an error occurs, **`globus_mutexattr_destroy()`** (p. 40) returns an implementation-specific non-zero error code.

#### 4.15.3.8 `int globus_rmutex_init (globus_rmutex_t * rmutex, globus_rmutexattr_t * rattr)`

Initialize a recursive mutex.

The **`globus_rmutex_init()`** (p. 40) function initializes a recursive mutex, that is, one which may be locked multiple times by a single thread without causing deadlock.

##### Parameters:

*rmutex* A pointer to the mutex to initialize

*rattr* IGNORED

##### Returns:

On success, **`globus_rmutex_init()`** (p. 40) initializes the mutex and returns `GLOBUS_SUCCESS`; otherwise, it returns a non-zero error code.

#### 4.15.3.9 `int globus_rmutex_lock (globus_rmutex_t * rmutex)`

Lock a recursive mutex.

The **`globus_rmutex_lock()`** (p. 40) function acquires the lock controlled by *rmutex*. This may be called multiple times in a single thread without causing deadlock, provided that a call to **`globus_rmutex_unlock()`** (p. 40) is called the same number of times as **`globus_rmutex_lock()`** (p. 40). Once acquired, all other threads calling this function will be blocked until the mutex is completely unlocked.

**Parameters:**

*rmutex* A pointer to the mutex to lock

**Returns:**

On success, **globus\_rmutex\_init()** (p. 40) increases the lock level for the mutex, blocks other threads trying to acquire the same mutex, and returns GLOBUS\_SUCCESS; otherwise, it returns a non-zero error code.

**4.15.3.10 int globus\_rmutex\_unlock (globus\_rmutex\_t \* rmutex)**

Unlock a recursive mutex.

The **globus\_rmutex\_unlock()** (p. 40) function decrements the lock count for the lock pointed to by *rmutex*. If the lock count is reduced to zero, it also unblocks a thread which is trying to acquire the lock if there is one.

**Parameters:**

*rmutex* Mutex to unlock

**Returns:**

GLOBUS\_SUCCESS

**4.15.3.11 int globus\_rmutex\_destroy (globus\_rmutex\_t \* rmutex)**

Destroy a recursive mutex.

The **globus\_rmutex\_destroy()** (p. 41) function destroys a recursive mutex. If the mutex is currently locked, behavior is undefined.

**Parameters:**

*rmutex* Mutex to unlock

**Returns:**

GLOBUS\_SUCCESS

## 4.16 Condition Variables

Collaboration diagram for Condition Variables:

**Data Structures**

- union **globus\_cond\_t**  
*Condition variable.*
- union **globus\_condattr\_t**  
*Condition variable attribute.*

## Functions

- `int globus_cond_init (globus_cond_t *cond, globus_condattr_t *attr)`
- `int globus_cond_destroy (globus_cond_t *cond)`
- `int globus_cond_wait (globus_cond_t *cond, globus_mutex_t *mutex)`
- `int globus_cond_timedwait (globus_cond_t *cond, globus_mutex_t *mutex, globus_abstime_t *abstime)`
- `int globus_cond_signal (globus_cond_t *cond)`
- `int globus_cond_broadcast (globus_cond_t *cond)`
- `int globus_condattr_init (globus_condattr_t *cond_attr)`
- `int globus_condattr_destroy (globus_condattr_t *cond_attr)`
- `int globus_condattr_setspace (globus_condattr_t *cond_attr, int space)`
- `int globus_condattr_getspace (globus_condattr_t *cond_attr, int *space)`

### 4.16.1 Detailed Description

The `globus_cond_t` (p. 52) provides condition variables for signalling events between threads interested in particular state. One or many threads may wait on a condition variable until it is signalled, at which point they can attempt to lock a mutex related to that condition's state and process the event.

In a non-threaded model, the condition variable wait operations are used to poll the event driver to handle any operations that have been scheduled for execution by the `globus_callback` system or I/O system. In this way, applications written to use those systems to handle nonblocking operations will work with either a threaded or nonthreaded runtime choice.

### 4.16.2 Function Documentation

#### 4.16.2.1 `int globus_cond_init (globus_cond_t * cond, globus_condattr_t * attr)`

Initialize a condition variable

The `globus_cond_init()` (p. 42) function creates a condition variable that can be used for event signalling between threads.

#### Parameters:

*cond* Pointer to the condition variable to initialize.

*attr* Condition variable attributes.

#### Returns:

On success, `globus_cond_init()` (p. 42) initializes the condition variable and returns `GLOBUS_SUCCESS`. Otherwise, a non-0 value is returned.

#### 4.16.2.2 `int globus_cond_destroy (globus_cond_t * cond)`

Destroy a condition variable.

The `globus_cond_destroy()` (p. 42) function destroys the condition variable pointed to by its *cond* parameter. After a condition variable is destroyed it may no longer be used unless it is again initialized by `globus_cond_init()` (p. 42).

#### Parameters:

*cond* The condition variable to destroy.

**Returns:**

On success, **globus\_cond\_destroy()** (p.42) returns GLOBUS\_SUCCESS. Otherwise, a non-zero implementation-specific error value is returned.

**4.16.2.3 int globus\_cond\_wait (globus\_cond\_t \* *cond*, globus\_mutex\_t \* *mutex*)**

Wait for a condition to be signalled.

The **globus\_cond\_wait()** (p.42) function atomically unlocks the mutex pointed to by the *mutex* parameter and blocks the current thread until the condition variable pointed to by *cond* is signalled by either **globus\_cond\_signal()** (p.43) or **globus\_cond\_broadcast()** (p.43). Behavior is undefined if **globus\_cond\_wait()** (p.42) is called with the mutex pointed to by the *mutex* variable unlocked.

**Parameters:**

*cond* The condition variable to wait for.

*mutex* The mutex associated with the condition state.

**Returns:**

On success, **globus\_cond\_wait()** (p.42) unlocks the mutex and blocks the current thread until it has been signalled, returning GLOBUS\_SUCCESS. Otherwise, **globus\_cond\_wait()** (p.42) returns an implementation-specific non-zero error value.

**4.16.2.4 int globus\_cond\_timedwait (globus\_cond\_t \* *cond*, globus\_mutex\_t \* *mutex*, globus\_abstime\_t \* *abstime*)**

Wait for a condition to be signalled.

The **globus\_cond\_timedwait()** (p.43) function atomically unlocks the mutex pointed to by the *mutex* parameter and blocks the current thread until either the condition variable pointed to by *cond* is signalled by another thread or the current time exceeds the value pointed to by the *abstime* parameter. If the timeout occurs before the condition is signalled, **globus\_cond\_timedwait()** (p.43) returns ETIMEDOUT. Behavior is undefined if **globus\_cond\_timedwait()** (p.43) is called with the mutex pointed to by the *mutex* variable unlocked.

**Parameters:**

*cond* The condition variable to wait for.

*mutex* The mutex associated with the condition state.

*abstime* The absolute time to wait until.

**Returns:**

On success, **globus\_cond\_timedwait()** (p.43) unlocks the mutex and blocks the current thread until it has been signalled, returning GLOBUS\_SUCCESS. If a timeout occurs before signal, **globus\_cond\_timedwait()** (p.43) unlocks the mutex and returns ETIMEDOUT. Otherwise, **globus\_cond\_timedwait()** (p.43) returns an implementation-specific non-zero error value.

**4.16.2.5 int globus\_cond\_signal (globus\_cond\_t \* *cond*)**

Signal a condition to a thread.

The **globus\_cond\_signal()** (p.43) function signals a condition as occurring. This will unblock at least one thread waiting for that condition.

**Parameters:**

*cond* A pointer to the condition variable to signal.

**Returns:**

Upon success, **globus\_cond\_signal()** (p. 43) returns GLOBUS\_SUCCESS. If an error occurs, **globus\_cond\_signal()** (p. 43) returns an implementation-specific non-zero error code.

#### 4.16.2.6 **int globus\_cond\_broadcast (globus\_cond\_t \* cond)**

Signal a condition to multiple threads.

The **globus\_cond\_signal()** (p. 43) function signals a condition as occurring. This will unblock all threads waiting for that condition.

**Parameters:**

*cond* A pointer to the condition variable to signal.

**Returns:**

Upon success, **globus\_cond\_broadcast()** (p. 43) returns GLOBUS\_SUCCESS. If an error occurs, **globus\_cond\_broadcast()** (p. 43) returns an implementation-specific non-zero error code.

#### 4.16.2.7 **int globus\_condattr\_init (globus\_condattr\_t \* cond\_attr)**

Initialize a condition variable attribute.

The **globus\_condattr\_init()** (p. 44) function initializes the condition variable attribute structure pointed to by its *cond\_attr* parameter to the system default values.

**Parameters:**

*cond\_attr* Attribute structure to initialize.

**Returns:**

Upon success, **globus\_condattr\_init()** (p. 44) returns GLOBUS\_SUCCESS and modifies the attribute pointed to by *cond\_attr*. If an error occurs, **globus\_condattr\_init()** (p. 44) returns an implementation-specific non-zero error code.

#### 4.16.2.8 **int globus\_condattr\_destroy (globus\_condattr\_t \* cond\_attr)**

Destroy a condition attribute.

The **globus\_condattr\_destroy()** (p. 44) function destroys the condition variable attribute structure pointed to by its *cond\_attr* parameter.

**Parameters:**

*cond\_attr* Attribute structure to destroy.

**Returns:**

Upon success, **globus\_condattr\_destroy()** (p. 44) returns GLOBUS\_SUCCESS and modifies the attribute pointed to by *cond\_attr*. If an error occurs, **globus\_condattr\_destroy()** (p. 44) returns an implementation-specific non-zero error code.

#### 4.16.2.9 `int globus_condattr_setspace (globus_condattr_t * cond_attr, int space)`

Set callback space associated with a condition variable attribute

The **`globus_condattr_setspace()`** (p. 44) function sets the callback space to use with condition variables created with this attribute.

Callback spaces are used to control how callbacks are issued to different threads. See **Callback Spaces** (p. 12) for more information on callback spaces.

##### Parameters:

*cond\_attr* Condition variable attribute to modify.

*space* Callback space to associate with the attribute.

##### Returns:

On success, **`globus_condattr_setspace()`** (p. 44) returns `GLOBUS_SUCCESS` and adds a reference to the callback space to the condition variable attribute. If an error occurs, **`globus_condattr_setspace()`** (p. 44) returns an implementation-specific non-zero error code.

#### 4.16.2.10 `int globus_condattr_getspace (globus_condattr_t * cond_attr, int * space)`

Get callback space associated with a condition variable attribute

The **`globus_condattr_getspace()`** (p. 45) function copies the value of the callback space associated with a condition variable attribute to the integer pointed to by the *space* parameter.

##### Parameters:

*cond\_attr* Condition variable attribute to modify.

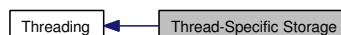
*space* Pointer to an integer to be set to point to the callback space associated with *cond\_attr*.

##### Returns:

On success, **`globus_condattr_getspace()`** (p. 45) returns `GLOBUS_SUCCESS` and modifies the value pointed to by *space* to refer to the callback space associated with *cond\_attr*. If an error occurs, **`globus_condattr_getspace()`** (p. 45) returns an implementation-specific non-zero error code.

## 4.17 Thread-Specific Storage

Collaboration diagram for Thread-Specific Storage:



### Functions

- `int globus_thread_key_create (globus_thread_key_t *key, globus_thread_key_destructor_func_t destructor)`
- `int globus_thread_key_delete (globus_thread_key_t key)`
- `void * globus_thread_getspecific (globus_thread_key_t key)`
- `int globus_thread_setspecific (globus_thread_key_t key, void *value)`

### 4.17.1 Detailed Description

The **globus\_thread\_key\_t** (p. 53) data type acts as a key to thread-specific storage. For each key created by **globus\_thread\_key\_create()** (p. 45), each thread may store and retrieve its own value.

### 4.17.2 Function Documentation

#### 4.17.2.1 **int globus\_thread\_key\_create (globus\_thread\_key\_t \* *key*, globus\_thread\_key\_destructor\_func\_t *destructor*)**

Create a key for thread-specific storage.

The **globus\_thread\_key\_create()** (p. 45) function creates a new key for thread-specific data. The new key will be available for all threads to store a distinct value. If the function pointer *destructor* is non-NULL, then that function will be invoked when a thread exits that has a non-NULL value associated with the key.

##### Parameters:

*key* Pointer to be set to the new key.

*destructor* Pointer to a function to call when a thread exits to free the key's value.

##### Returns:

On success, **globus\_thread\_create\_key()** will create a new key to thread-local storage and return GLOBUS\_SUCCESS. If an error occurs, then the value of *key* is undefined and **globus\_thread\_create\_key()** returns an implementation-specific non-zero error value.

#### 4.17.2.2 **int globus\_thread\_key\_delete (globus\_thread\_key\_t *key*)**

Delete a thread-local storage key.

The **globus\_thread\_key\_delete()** (p. 46) function deletes the key used for a thread-local storage association. The destructor function for this key will no longer be called after this function returns. The behavior of subsequent calls to **globus\_thread\_getspecific()** (p. 46) or **globus\_thread\_setspecific()** (p. 46) with this key will be undefined.

##### Parameters:

*key* Key to destroy.

##### Returns:

On success, **globus\_thread\_key\_delete()** (p. 46) will delete a thread-local storage key and return GLOBUS\_SUCCESS. If an error occurs, then the value of *key* is undefined and **globus\_thread\_create\_key()** returns an implementation-specific non-zero error value.

#### 4.17.2.3 **void\* globus\_thread\_getspecific (globus\_thread\_key\_t *key*)**

Get a thread-specific data value.

The **globus\_thread\_getspecific()** (p. 46) function returns the value associated with the thread-specific data key passed as its first parameter. This function returns NULL if the value has not been set by the current thread. The return value is undefined if the key is not valid.

##### Parameters:

*key* Thread-specific data key to look up.

##### Returns:

The value passed to a previous call to **globus\_thread\_setspecific()** (p. 46) in the current thread for this key.

#### 4.17.2.4 `int globus_thread_setspecific (globus_thread_key_t key, void * value)`

Set a thread-specific data value.

The `globus_thread_setspecific()` (p. 46) function associates a thread-specific value with a data key. If the key had a previous value set in the current thread, it is replaced, but the destructor function is not called for the old value.

##### Parameters:

*key* Thread-specific data key to store.

*value* A pointer to data to store as the thread-specific data for this thread.

##### Returns:

On success, `globus_thread_setspecific()` (p. 46) stores value in the thread-specific data for the specified key and returns `GLOBUS_SUCCESS`. If an error occurs, `globus_thread_setspecific()` (p. 46) returns an implementation-specific non-zero error code and does not modify the key's value for this thread.

## 4.18 One-time execution

Collaboration diagram for One-time execution:



### Data Structures

- union `globus_thread_once_t`  
*Thread once structure.*

### Defines

- `#define GLOBUS_THREAD_ONCE_INIT { .none = 0 }`

### Functions

- `int globus_thread_once (globus_thread_once_t *once, void(*init_routine)(void))`

#### 4.18.1 Detailed Description

The `globus_thread_once_t` (p. 53) provides a way for applications and libraries to execute some code exactly one time, independent of the number of threads which attempt to execute it. To use this, statically initialize a `globus_thread_once_t` (p. 53) control with the value `GLOBUS_THREAD_ONCE_INIT`, and pass a pointer to a function to execute once, along with the control, to `globus_thread_once()` (p. 47).

#### 4.18.2 Define Documentation

##### 4.18.2.1 `#define GLOBUS_THREAD_ONCE_INIT { .none = 0 }`

Thread once initializer value.



### 4.18.3 Function Documentation

#### 4.18.3.1 `int globus_thread_once (globus_thread_once_t * once, void(*) (void) init_routine)`

Execute a function one time.

The `globus_thread_once()` (p. 47) function will execute the function pointed to by its *init\_routine* parameter one time for each unique `globus_thread_once_t` (p. 53) object passed to it, independent of the number of threads calling it. The *once* value must be a static value initialized to `GLOBUS_THREAD_ONCE_INIT`.

#### Parameters:

*once* A pointer to the value used to govern whether the function passed via the *init\_routine* parameter has executed.

*init\_routine* Function to execute one time. It is called with no parameters.

#### Returns:

On success, `globus_thread_once()` (p. 47) guarantees that the function pointed to by *init\_routine* has run, and that subsequent calls to `globus_thread_once()` (p. 47) with the same value of *once* will not execute that function, and returns `GLOBUS_SUCCESS`. If an error occurs, `globus_thread_once()` (p. 47) returns an implementation-specific non-zero error value.

## 4.19 URL String Parser

The Globus URL functions provide a simple mechanism for parsing a URL string into a data structure, and for determining the scheme of an URL string.

### Data Structures

- struct `globus_url_t`  
*Parsed URLs.*

### Enumerations

- enum `globus_url_scheme_t` {  
    `GLOBUS_URL_SCHEME_FTP = 0`,  
    `GLOBUS_URL_SCHEME_GSIFTP`,  
    `GLOBUS_URL_SCHEME_HTTP`,  
    `GLOBUS_URL_SCHEME_HTTPS`,  
    `GLOBUS_URL_SCHEME_LDAP`,  
    `GLOBUS_URL_SCHEME_FILE`,  
    `GLOBUS_URL_SCHEME_X_NEXUS`,  
    `GLOBUS_URL_SCHEME_X_GASS_CACHE`,  
    `GLOBUS_URL_SCHEME_UNKNOWN`,  
    `GLOBUS_URL_NUM_SCHEMES` }

## Functions

- `int globus_url_parse (const char *url_string, globus_url_t *url)`
- `int globus_url_parse_rfc1738 (const char *url_string, globus_url_t *url)`
- `int globus_url_parse_loose (const char *url_string, globus_url_t *url)`
- `int globus_url_destroy (globus_url_t *url)`
- `int globus_url_get_scheme (const char *url_string, globus_url_scheme_t *scheme_type)`
- `int globus_url_copy (globus_url_t *dst, const globus_url_t *src)`

### 4.19.1 Detailed Description

The Globus URL functions provide a simple mechanism for parsing a URL string into a data structure, and for determining the scheme of an URL string.

These functions are part of the Globus common library. The GLOBUS\_COMMON module must be activated in order to use them.

### 4.19.2 Enumeration Type Documentation

#### 4.19.2.1 enum globus\_url\_scheme\_t

URL Schemes.

The Globus URL library supports a set of URL schemes (protocols). This enumeration can be used to quickly dispatch a parsed URL based on a constant value.

See also:

`globus_url_t::scheme_type` (p. 54)

Enumerator:

**GLOBUS\_URL\_SCHEME\_FTP** File Transfer Protocol.

**GLOBUS\_URL\_SCHEME\_GSIFTP** GSI-enhanced File Transfer Protocol.

**GLOBUS\_URL\_SCHEME\_HTTP** HyperText Transfer Protocol.

**GLOBUS\_URL\_SCHEME\_HTTPS** Secure HyperText Transfer Protocol.

**GLOBUS\_URL\_SCHEME\_LDAP** Lightweight Directory Access Protocol.

**GLOBUS\_URL\_SCHEME\_FILE** File Location.

**GLOBUS\_URL\_SCHEME\_X\_NEXUS** Nexus endpoint.

**GLOBUS\_URL\_SCHEME\_X\_GASS\_CACHE** GASS Cache Entry.

**GLOBUS\_URL\_SCHEME\_UNKNOWN** Any other URL of the form **scheme://something**.

**GLOBUS\_URL\_NUM\_SCHEMES** Total number of URL schemes supported.

### 4.19.3 Function Documentation

#### 4.19.3.1 int globus\_url\_parse (const char \* url\_string, globus\_url\_t \* url)

Parse a string containing a URL into a `globus_url_t` (p. 54).

Parameters:

*url\_string* String to parse

*url* Pointer to `globus_url_t` (p. 54) to be filled with the fields of the url

**Return values:**

**GLOBUS\_SUCCESS** The string was successfully parsed.

**GLOBUS\_URL\_ERROR\_NULL\_STRING** The url\_string was GLOBUS\_NULL.

**GLOBUS\_URL\_ERROR\_NULL\_URL** The URL pointer was GLOBUS\_NULL.

**GLOBUS\_URL\_ERROR\_BAD\_SCHEME** The URL scheme (protocol) contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_USER** The user part of the URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_PASSWORD** The password part of the URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_HOST** The host part of the URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_PORT** The port part of the URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_PATH** The path part of the URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_DN** -9 The DN part of an LDAP URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_ATTRIBUTES** -10 The attributes part of an LDAP URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_SCOPE** -11 The scope part of an LDAP URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_FILTER** -12 The filter part of an LDAP URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_OUT\_OF\_MEMORY** -13 The library was unable to allocate memory to create the the **globus\_url\_t** (p. 54) contents.

**GLOBUS\_URL\_ERROR\_INTERNAL\_ERROR** -14 Some unexpected error occurred parsing the URL.

**4.19.3.2 int globus\_url\_parse\_rfc1738 (const char \* url\_string, globus\_url\_t \* url)**

Parse a string containing a URL into a **globus\_url\_t** (p. 54).

**Parameters:**

**url\_string** String to parse

**url** Pointer to **globus\_url\_t** (p. 54) to be filled with the fields of the url

**Return values:**

**GLOBUS\_SUCCESS** The string was successfully parsed.

**GLOBUS\_URL\_ERROR\_NULL\_STRING** The url\_string was GLOBUS\_NULL.

**GLOBUS\_URL\_ERROR\_NULL\_URL** The URL pointer was GLOBUS\_NULL.

**GLOBUS\_URL\_ERROR\_BAD\_SCHEME** The URL scheme (protocol) contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_USER** The user part of the URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_PASSWORD** The password part of the URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_HOST** The host part of the URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_PORT** The port part of the URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_PATH** The path part of the URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_DN** -9 The DN part of an LDAP URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_ATTRIBUTES** -10 The attributes part of an LDAP URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_SCOPE** -11 The scope part of an LDAP URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_FILTER** -12 The filter part of an LDAP URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_OUT\_OF\_MEMORY** -13 The library was unable to allocate memory to create the the **globus\_url\_t** (p. 54) contents.

**GLOBUS\_URL\_ERROR\_INTERNAL\_ERROR** -14 Some unexpected error occurred parsing the URL.

#### 4.19.3.3 **int globus\_url\_parse\_loose (const char \* *url\_string*, globus\_url\_t \* *url*)**

Parse a string containing a URL into a **globus\_url\_t** (p. 54) Looser restrictions on characters allowed in the path part of the URL.

##### **Parameters:**

*url\_string* String to parse

*url* Pointer to **globus\_url\_t** (p. 54) to be filled with the fields of the url

##### **Return values:**

**GLOBUS\_SUCCESS** The string was successfully parsed.

**GLOBUS\_URL\_ERROR\_NULL\_STRING** The *url\_string* was GLOBUS\_NULL.

**GLOBUS\_URL\_ERROR\_NULL\_URL** The URL pointer was GLOBUS\_NULL.

**GLOBUS\_URL\_ERROR\_BAD\_SCHEME** The URL scheme (protocol) contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_USER** The user part of the URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_PASSWORD** The password part of the URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_HOST** The host part of the URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_PORT** The port part of the URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_PATH** The path part of the URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_DN** -9 The DN part of an LDAP URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_ATTRIBUTES** -10 The attributes part of an LDAP URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_SCOPE** -11 The scope part of an LDAP URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_BAD\_FILTER** -12 The filter part of an LDAP URL contained invalid characters.

**GLOBUS\_URL\_ERROR\_OUT\_OF\_MEMORY** -13 The library was unable to allocate memory to create the the **globus\_url\_t** (p. 54) contents.

**GLOBUS\_URL\_ERROR\_INTERNAL\_ERROR** -14 Some unexpected error occurred parsing the URL.

#### 4.19.3.4 **int globus\_url\_destroy (globus\_url\_t \* *url*)**

Destroy a **globus\_url\_t** (p. 54) structure.

This function frees all memory associated with a **globus\_url\_t** (p. 54) structure.

##### **Parameters:**

*url* The url structure to destroy

##### **Return values:**

**GLOBUS\_SUCCESS** The URL was successfully destroyed.

#### 4.19.3.5 **int globus\_url\_get\_scheme (const char \* *url\_string*, globus\_url\_scheme\_t \* *scheme\_type*)**

Get the scheme of an URL.

This function determines the scheme type of the url string, and populates the variable pointed to by second parameter with that value. This performs a less expensive parsing than **globus\_url\_parse()** (p. 49) and is suitable for applications which need only to choose a handler based on the URL scheme.

**Parameters:**

*url\_string* The string containing the URL.

*scheme\_type* A pointer to a `globus_url_scheme_t` which will be set to the scheme.

**Return values:**

***GLOBUS\_SUCCESS*** The URL scheme was recognized, and `scheme_type` has been updated.

***GLOBUS\_URL\_ERROR\_BAD\_SCHEME*** The URL scheme was not recognized.

**4.19.3.6 int globus\_url\_copy (globus\_url\_t \* dst, const globus\_url\_t \* src)**

Create a copy of an URL structure.

This function copies the contents of a url structure into another.

**Parameters:**

*dst* The URL structure to be populated with a copy of the contents of `src`.

*src* The original URL.

**Return values:**

***GLOBUS\_SUCCESS*** The URL was successfully copied.

***GLOBUS\_URL\_ERROR\_NULL\_URL*** One of the URLs was `GLOBUS_NULL`.

***GLOBUS\_URL\_ERROR\_OUT\_OF\_MEMORY***; The library was unable to allocate memory to create the `globus_url_t` (p. 54) contents.

## 5 globus common Data Structure Documentation

### 5.1 globus\_cond\_t Union Reference

Condition variable.

#### 5.1.1 Detailed Description

Condition variable.

### 5.2 globus\_condattr\_t Union Reference

Condition variable attribute.

#### 5.2.1 Detailed Description

Condition variable attribute.

### 5.3 globus\_mutex\_t Union Reference

Mutex.

### **5.3.1 Detailed Description**

Mutex.

## **5.4 globus\_mutexattr\_t Union Reference**

Mutex attribute.

### **5.4.1 Detailed Description**

Mutex attribute.

## **5.5 globus\_rmutex\_t Struct Reference**

Recursive Mutex.

### **5.5.1 Detailed Description**

Recursive Mutex.

See also:

**globus\_rmutex\_init()** (p. 40), **globus\_rmutex\_destroy()** (p. 41), **globus\_rmutex\_lock()** (p. 40), **globus\_rmutex\_unlock()** (p. 40)

## **5.6 globus\_thread\_key\_t Union Reference**

Thread-specific data key.

### **5.6.1 Detailed Description**

Thread-specific data key.

## **5.7 globus\_thread\_once\_t Union Reference**

Thread once structure.

### **5.7.1 Detailed Description**

Thread once structure.

## **5.8 globus\_thread\_t Union Reference**

Thread ID.

### **5.8.1 Detailed Description**

Thread ID.

## 5.9 globus\_threadattr\_t Union Reference

Thread attributes.

### 5.9.1 Detailed Description

Thread attributes.

## 5.10 globus\_url\_t Struct Reference

Parsed URLs.

### Data Fields

- `char * scheme`
- `globus_url_scheme_t scheme_type`
- `char * user`
- `char * password`
- `char * host`
- `unsigned short port`
- `char * url_path`
- `char * dn`
- `char * attributes`
- `char * scope`
- `char * filter`
- `char * url_specific_part`

### 5.10.1 Detailed Description

Parsed URLs.

This structure contains the fields which were parsed from a string representation of a URL. There are no methods to access fields of this structure.

### 5.10.2 Field Documentation

#### 5.10.2.1 `char* globus_url_t::scheme`

A string containing the URL's scheme (http, ftp, etc).

#### 5.10.2.2 `globus_url_scheme_t globus_url_t::scheme_type`

An enumerated scheme type.

This is derived from the scheme string

#### 5.10.2.3 `char* globus_url_t::user`

The username portion of the URL.

[ftp, gsiftp]

#### **5.10.2.4 char\* globus\_url\_t::password**

The user's password from the URL.

[ftp, gsiftp]

#### **5.10.2.5 char\* globus\_url\_t::host**

The host name or IP address of the URL.

[ftp, gsiftp, http, https, ldap, x-nexus]

#### **5.10.2.6 unsigned short globus\_url\_t::port**

The TCP port number of the service providing the URL [ftp, gsiftp, http, https, ldap, x-nexus].

#### **5.10.2.7 char\* globus\_url\_t::url\_path**

The path name of the resource on the service providing the URL.

[ftp, gsiftp, http, https]

#### **5.10.2.8 char\* globus\_url\_t::dn**

The distinguished name for the base of an LDAP search.

[ldap]

#### **5.10.2.9 char\* globus\_url\_t::attributes**

The list of attributes which should be returned from an LDAP search.

[ldap]

#### **5.10.2.10 char\* globus\_url\_t::scope**

The scope of an LDAP search.

[ldap]

#### **5.10.2.11 char\* globus\_url\_t::filter**

The filter to be applied to an LDAP search [ldap].

#### **5.10.2.12 char\* globus\_url\_t::url\_specific\_part**

An unparsed string containing the remaining text after the optional host and port of an unknown URL, or the contents of a x-gass-cache URL [x-gass-cache, unknown].

## **6 globus common Page Documentation**

### **6.1 Deprecated List**

Global GLOBUS\_POLL\_MODULE (p. 3)



## Index

- attributes
  - globus\_url\_t, 55
- Condition Variables, 41
- dn
  - globus\_url\_t, 55
- Error Construction, 18, 23
- Error Data Accessors and Modifiers, 20, 26
- Error Handling Helpers, 21, 29
- filter
  - globus\_url\_t, 55
- Globus Callback, 2
- Globus Callback API, 4
- Globus Callback Signal Handling, 16
- Globus Callback Spaces, 12
- Globus Errno Error API, 18
- Globus Error API, 22
- Globus Generic Error API, 23
- globus\_callback
  - GLOBUS\_CALLBACK\_ERROR\_ALREADY\_-CANCELED, 4
  - GLOBUS\_CALLBACK\_ERROR\_INVALID\_-ARGUMENT, 4
  - GLOBUS\_CALLBACK\_ERROR\_INVALID\_-CALLBACK\_HANDLE, 4
  - GLOBUS\_CALLBACK\_ERROR\_INVALID\_-SPACE, 4
  - GLOBUS\_CALLBACK\_ERROR\_MEMORY\_-ALLOC, 4
  - GLOBUS\_CALLBACK\_ERROR\_NO\_-ACTIVE\_CALLBACK, 4
  - globus\_callback\_error\_type\_t, 4
  - globus\_callback\_handle\_t, 3
  - GLOBUS\_CALLBACK\_MODULE, 3
  - globus\_callback\_space\_attr\_t, 3
  - globus\_callback\_space\_t, 3
  - GLOBUS\_POLL\_MODULE, 3
- globus\_callback\_add\_wakeup\_handler
  - globus\_callback\_signal, 17
- globus\_callback\_adjust\_oneshot
  - globus\_callback\_api, 9
- globus\_callback\_adjust\_period
  - globus\_callback\_api, 9
- globus\_callback\_api
  - globus\_callback\_adjust\_oneshot, 9
  - globus\_callback\_adjust\_period, 9
  - globus\_callback\_func\_t, 7
  - globus\_callback\_get\_timeout, 11
  - globus\_callback\_has\_time\_expired, 11
  - globus\_callback\_poll, 5
  - globus\_callback\_register\_oneshot, 6
  - globus\_callback\_register\_periodic, 6
  - globus\_callback\_register\_signal\_handler, 6
  - globus\_callback\_signal\_poll, 10
  - globus\_callback\_space\_poll, 10
  - globus\_callback\_space\_register\_oneshot, 7
  - globus\_callback\_space\_register\_periodic, 8
  - globus\_callback\_unregister, 8
  - globus\_callback\_was\_restarted, 11
  - globus\_poll, 6
  - globus\_poll\_blocking, 5
  - globus\_poll\_nonblocking, 5
  - globus\_signal\_poll, 6
- GLOBUS\_CALLBACK\_ERROR\_ALREADY\_-CANCELED
  - globus\_callback, 4
- GLOBUS\_CALLBACK\_ERROR\_INVALID\_-ARGUMENT
  - globus\_callback, 4
- GLOBUS\_CALLBACK\_ERROR\_INVALID\_-CALLBACK\_HANDLE
  - globus\_callback, 4
- GLOBUS\_CALLBACK\_ERROR\_INVALID\_SPACE
  - globus\_callback, 4
- GLOBUS\_CALLBACK\_ERROR\_MEMORY\_-ALLOC
  - globus\_callback, 4
- GLOBUS\_CALLBACK\_ERROR\_NO\_ACTIVE\_CALLBACK
  - globus\_callback, 4
- globus\_callback\_error\_type\_t
  - globus\_callback, 4
- globus\_callback\_func\_t
  - globus\_callback\_api, 7
- globus\_callback\_get\_timeout
  - globus\_callback\_api, 11
- GLOBUS\_CALLBACK\_GLOBAL\_SPACE
  - globus\_callback\_spaces, 12
- globus\_callback\_handle\_t
  - globus\_callback, 3
- globus\_callback\_has\_time\_expired
  - globus\_callback\_api, 11
- GLOBUS\_CALLBACK\_MODULE
  - globus\_callback, 3
- globus\_callback\_poll
  - globus\_callback\_api, 5
- globus\_callback\_register\_oneshot
  - globus\_callback\_api, 6
- globus\_callback\_register\_periodic
  - globus\_callback\_api, 6
- globus\_callback\_register\_signal\_handler
  - globus\_callback\_api, 6
- globus\_callback\_poll\_signal

- globus\_callback\_add\_wakeup\_handler, 17
- globus\_callback\_space\_register\_signal\_handler, 17
- globus\_callback\_unregister\_signal\_handler, 17
- GLOBUS\_SIGNAL\_INTERRUPT, 17
- globus\_callback\_signal\_poll
  - globus\_callback\_api, 10
- globus\_callback\_space\_attr\_destroy
  - globus\_callback\_spaces, 14
- globus\_callback\_space\_attr\_get\_behavior
  - globus\_callback\_spaces, 15
- globus\_callback\_space\_attr\_init
  - globus\_callback\_spaces, 14
- globus\_callback\_space\_attr\_set\_behavior
  - globus\_callback\_spaces, 15
- globus\_callback\_space\_attr\_t
  - globus\_callback, 3
- GLOBUS\_CALLBACK\_SPACE\_BEHAVIOR\_-SERIALIZED
  - globus\_callback\_spaces, 13
- GLOBUS\_CALLBACK\_SPACE\_BEHAVIOR\_-SINGLE
  - globus\_callback\_spaces, 13
- globus\_callback\_space\_behavior\_t
  - globus\_callback\_spaces, 12
- GLOBUS\_CALLBACK\_SPACE\_BEHAVIOR\_-THREADED
  - globus\_callback\_spaces, 13
- globus\_callback\_space\_destroy
  - globus\_callback\_spaces, 14
- globus\_callback\_space\_get
  - globus\_callback\_spaces, 15
- globus\_callback\_space\_get\_depth
  - globus\_callback\_spaces, 16
- globus\_callback\_space\_init
  - globus\_callback\_spaces, 13
- globus\_callback\_space\_is\_single
  - globus\_callback\_spaces, 16
- globus\_callback\_space\_poll
  - globus\_callback\_api, 10
- globus\_callback\_space\_reference
  - globus\_callback\_spaces, 13
- globus\_callback\_space\_register\_one\_shot
  - globus\_callback\_api, 7
- globus\_callback\_space\_register\_periodic
  - globus\_callback\_api, 8
- globus\_callback\_space\_register\_signal\_handler
  - globus\_callback\_signal, 17
- globus\_callback\_space\_t
  - globus\_callback, 3
- globus\_callback\_spaces
  - GLOBUS\_CALLBACK\_GLOBAL\_SPACE, 12
  - globus\_callback\_space\_attr\_destroy, 14
  - globus\_callback\_space\_attr\_get\_behavior, 15
  - globus\_callback\_space\_attr\_init, 14
  - globus\_callback\_space\_attr\_set\_behavior, 15

- GLOBUS\_CALLBACK\_SPACE\_BEHAVIOR\_-SERIALIZED, 13
- GLOBUS\_CALLBACK\_SPACE\_BEHAVIOR\_-SINGLE, 13
- globus\_callback\_space\_behavior\_t, 12
- GLOBUS\_CALLBACK\_SPACE\_BEHAVIOR\_-THREADED, 13
- globus\_callback\_space\_destroy, 14
- globus\_callback\_space\_get, 15
- globus\_callback\_space\_get\_depth, 16
- globus\_callback\_space\_init, 13
- globus\_callback\_space\_is\_single, 16
- globus\_callback\_space\_reference, 13
- globus\_callback\_unregister
  - globus\_callback\_api, 8
- globus\_callback\_unregister\_signal\_handler
  - globus\_callback\_signal, 17
- globus\_callback\_was\_restarted
  - globus\_callback\_api, 11
- globus\_cond
  - globus\_cond\_broadcast, 43
  - globus\_cond\_destroy, 42
  - globus\_cond\_init, 42
  - globus\_cond\_signal, 43
  - globus\_cond\_timedwait, 43
  - globus\_cond\_wait, 42
  - globus\_condattr\_destroy, 44
  - globus\_condattr\_getspace, 44
  - globus\_condattr\_init, 44
  - globus\_condattr\_setspace, 44
- globus\_cond\_broadcast
  - globus\_cond, 43
- globus\_cond\_destroy
  - globus\_cond, 42
- globus\_cond\_init
  - globus\_cond, 42
- globus\_cond\_signal
  - globus\_cond, 43
- globus\_cond\_t, 52
- globus\_cond\_timedwait
  - globus\_cond, 43
- globus\_cond\_wait
  - globus\_cond, 42
- globus\_condattr\_destroy
  - globus\_cond, 44
- globus\_condattr\_getspace
  - globus\_cond, 44
- globus\_condattr\_init
  - globus\_cond, 44
- globus\_condattr\_setspace
  - globus\_cond, 44
- globus\_condattr\_t, 52
- globus\_errno\_error\_accessor
  - globus\_error\_errno\_get\_errno, 20
  - globus\_error\_errno\_set\_errno, 20
- globus\_errno\_error\_object

- globus\_error\_construct\_errno\_error, 19
- globus\_error\_initialize\_errno\_error, 19
- GLOBUS\_ERROR\_TYPE\_ERRNO, 19
- globus\_errno\_error\_utility
  - globus\_error\_errno\_match, 21
  - globus\_error\_wrap\_errno\_error, 21
- globus\_error\_construct\_errno\_error
  - globus\_errno\_error\_object, 19
- globus\_error\_construct\_error
  - globus\_generic\_error\_object, 24
- globus\_error\_errno\_get\_errno
  - globus\_errno\_error\_accessor, 20
- globus\_error\_errno\_match
  - globus\_errno\_error\_utility, 21
- globus\_error\_errno\_set\_errno
  - globus\_errno\_error\_accessor, 20
- globus\_error\_get\_cause
  - globus\_generic\_error\_accessor, 27
- globus\_error\_get\_long\_desc
  - globus\_generic\_error\_accessor, 28
- globus\_error\_get\_short\_desc
  - globus\_generic\_error\_accessor, 28
- globus\_error\_get\_source
  - globus\_generic\_error\_accessor, 27
- globus\_error\_get\_type
  - globus\_generic\_error\_accessor, 27
- globus\_error\_initialize\_errno\_error
  - globus\_errno\_error\_object, 19
- globus\_error\_initialize\_error
  - globus\_generic\_error\_object, 25
- globus\_error\_match
  - globus\_generic\_error\_utility, 30
- globus\_error\_print\_chain
  - globus\_generic\_error\_utility, 30
- globus\_error\_print\_friendly
  - globus\_generic\_error\_utility, 30
- globus\_error\_set\_cause
  - globus\_generic\_error\_accessor, 27
- globus\_error\_set\_long\_desc
  - globus\_generic\_error\_accessor, 29
- globus\_error\_set\_short\_desc
  - globus\_generic\_error\_accessor, 28
- globus\_error\_set\_source
  - globus\_generic\_error\_accessor, 27
- globus\_error\_set\_type
  - globus\_generic\_error\_accessor, 28
- GLOBUS\_ERROR\_TYPE\_ERRNO
  - globus\_errno\_error\_object, 19
- GLOBUS\_ERROR\_TYPE\_GLOBUS
  - globus\_generic\_error\_object, 24
- globus\_error\_v\_construct\_error
  - globus\_generic\_error\_object, 25
- globus\_error\_wrap\_errno\_error
  - globus\_errno\_error\_utility, 21
- globus\_generic\_error\_accessor
  - globus\_error\_get\_cause, 27
- globus\_error\_get\_long\_desc, 28
- globus\_error\_get\_short\_desc, 28
- globus\_error\_get\_source, 27
- globus\_error\_get\_type, 27
- globus\_error\_set\_cause, 27
- globus\_error\_set\_long\_desc, 29
- globus\_error\_set\_short\_desc, 28
- globus\_error\_set\_source, 27
- globus\_error\_set\_type, 28
- globus\_generic\_error\_object
  - globus\_error\_construct\_error, 24
  - globus\_error\_initialize\_error, 25
  - GLOBUS\_ERROR\_TYPE\_GLOBUS, 24
  - globus\_error\_v\_construct\_error, 25
- globus\_generic\_error\_utility
  - globus\_error\_match, 30
  - globus\_error\_print\_chain, 30
  - globus\_error\_print\_friendly, 30
- globus\_i\_am\_only\_thread
  - globus\_thread, 35
- globus\_mutex
  - globus\_mutex\_destroy, 38
  - globus\_mutex\_init, 38
  - globus\_mutex\_lock, 38
  - globus\_mutex\_trylock, 39
  - globus\_mutex\_unlock, 38
  - globus\_mutexattr\_destroy, 39
  - globus\_mutexattr\_init, 39
  - globus\_rmutex\_destroy, 41
  - globus\_rmutex\_init, 40
  - globus\_rmutex\_lock, 40
  - globus\_rmutex\_unlock, 40
  - globus\_rmutexattr\_t, 37
- globus\_mutex\_destroy
  - globus\_mutex, 38
- globus\_mutex\_init
  - globus\_mutex, 38
- globus\_mutex\_lock
  - globus\_mutex, 38
- globus\_mutex\_t, 52
- globus\_mutex\_trylock
  - globus\_mutex, 39
- globus\_mutex\_unlock
  - globus\_mutex, 38
- globus\_mutexattr\_destroy
  - globus\_mutex, 39
- globus\_mutexattr\_init
  - globus\_mutex, 39
- globus\_mutexattr\_t, 52
- globus\_poll
  - globus\_callback\_api, 6
- globus\_poll\_blocking
  - globus\_callback\_api, 5
- GLOBUS\_POLL\_MODULE
  - globus\_callback, 3
- globus\_poll\_nonblocking

- globus\_callback\_api, 5
- globus\_rmutex\_destroy
  - globus\_mutex, 41
- globus\_rmutex\_init
  - globus\_mutex, 40
- globus\_rmutex\_lock
  - globus\_mutex, 40
- globus\_rmutex\_t, 53
- globus\_rmutex\_unlock
  - globus\_mutex, 40
- globus\_rmutexattr\_t
  - globus\_mutex, 37
- GLOBUS\_SIGNAL\_INTERRUPT
  - globus\_callback\_signal, 17
- globus\_signal\_poll
  - globus\_callback\_api, 6
- globus\_thread
  - globus\_i\_am\_only\_thread, 35
  - globus\_thread\_cancel, 35
  - GLOBUS\_THREAD\_CANCEL\_DISABLE, 32
  - GLOBUS\_THREAD\_CANCEL\_ENABLE, 32
  - globus\_thread\_cancellable\_func, 35
  - globus\_thread\_create, 33
  - globus\_thread\_equal, 35
  - globus\_thread\_exit, 34
  - globus\_thread\_key\_destructor\_func\_t, 33
  - globus\_thread\_kill, 34
  - GLOBUS\_THREAD\_MODULE, 32
  - globus\_thread\_preemptive\_threads, 35
  - globus\_thread\_self, 34
  - globus\_thread\_set\_model, 33
  - globus\_thread\_setcancelstate, 36
  - globus\_thread\_sigmask, 34
  - globus\_thread\_testcancel, 36
  - globus\_thread\_yield, 33
- globus\_thread\_cancel
  - globus\_thread, 35
- GLOBUS\_THREAD\_CANCEL\_DISABLE
  - globus\_thread, 32
- GLOBUS\_THREAD\_CANCEL\_ENABLE
  - globus\_thread, 32
- globus\_thread\_cancellable\_func
  - globus\_thread, 35
- globus\_thread\_create
  - globus\_thread, 33
- globus\_thread\_equal
  - globus\_thread, 35
- globus\_thread\_exit
  - globus\_thread, 34
- globus\_thread\_getspecific
  - globus\_thread\_key, 46
- globus\_thread\_key
  - globus\_thread\_getspecific, 46
  - globus\_thread\_key\_create, 45
  - globus\_thread\_key\_delete, 46
  - globus\_thread\_setspecific, 46
- globus\_thread\_key\_create
  - globus\_thread\_key, 45
- globus\_thread\_key\_delete
  - globus\_thread\_key, 46
- globus\_thread\_key\_destructor\_func\_t
  - globus\_thread, 33
- globus\_thread\_key\_t, 53
- globus\_thread\_kill
  - globus\_thread, 34
- GLOBUS\_THREAD\_MODULE
  - globus\_thread, 32
- globus\_thread\_once
  - globus\_thread\_once, 47
  - GLOBUS\_THREAD\_ONCE\_INIT, 47
- GLOBUS\_THREAD\_ONCE\_INIT
  - globus\_thread\_once, 47
- globus\_thread\_once\_t, 53
- globus\_thread\_preemptive\_threads
  - globus\_thread, 35
- globus\_thread\_self
  - globus\_thread, 34
- globus\_thread\_set\_model
  - globus\_thread, 33
- globus\_thread\_setcancelstate
  - globus\_thread, 36
- globus\_thread\_setspecific
  - globus\_thread\_key, 46
- globus\_thread\_sigmask
  - globus\_thread, 34
- globus\_thread\_t, 53
- globus\_thread\_testcancel
  - globus\_thread, 36
- globus\_thread\_yield
  - globus\_thread, 33
- globus\_threadattr\_t, 53
- globus\_url
  - globus\_url\_copy, 51
  - globus\_url\_destroy, 51
  - globus\_url\_get\_scheme, 51
  - GLOBUS\_URL\_NUM\_SCHEMES, 49
  - globus\_url\_parse, 49
  - globus\_url\_parse\_loose, 50
  - globus\_url\_parse\_rfc1738, 50
  - GLOBUS\_URL\_SCHEME\_FILE, 49
  - GLOBUS\_URL\_SCHEME\_FTP, 49
  - GLOBUS\_URL\_SCHEME\_GSIFTP, 49
  - GLOBUS\_URL\_SCHEME\_HTTP, 49
  - GLOBUS\_URL\_SCHEME\_HTTPS, 49
  - GLOBUS\_URL\_SCHEME\_LDAP, 49
  - globus\_url\_scheme\_t, 49
  - GLOBUS\_URL\_SCHEME\_UNKNOWN, 49
  - GLOBUS\_URL\_SCHEME\_X\_GASS\_CACHE, 49
  - GLOBUS\_URL\_SCHEME\_X\_NEXUS, 49
- globus\_url\_copy
  - globus\_url, 51

- globus\_url\_destroy
  - globus\_url, 51
- globus\_url\_get\_scheme
  - globus\_url, 51
- GLOBUS\_URL\_NUM\_SCHEMES
  - globus\_url, 49
- globus\_url\_parse
  - globus\_url, 49
- globus\_url\_parse\_loose
  - globus\_url, 50
- globus\_url\_parse\_rfc1738
  - globus\_url, 50
- GLOBUS\_URL\_SCHEME\_FILE
  - globus\_url, 49
- GLOBUS\_URL\_SCHEME\_FTP
  - globus\_url, 49
- GLOBUS\_URL\_SCHEME\_GSIFTP
  - globus\_url, 49
- GLOBUS\_URL\_SCHEME\_HTTP
  - globus\_url, 49
- GLOBUS\_URL\_SCHEME\_HTTPS
  - globus\_url, 49
- GLOBUS\_URL\_SCHEME\_LDAP
  - globus\_url, 49
- globus\_url\_scheme\_t
  - globus\_url, 49
- GLOBUS\_URL\_SCHEME\_UNKNOWN
  - globus\_url, 49
- GLOBUS\_URL\_SCHEME\_X\_GASS\_CACHE
  - globus\_url, 49
- GLOBUS\_URL\_SCHEME\_X\_NEXUS
  - globus\_url, 49
- globus\_url\_t, 54
  - attributes, 55
  - dn, 55
  - filter, 55
  - host, 54
  - password, 54
  - port, 54
  - scheme, 54
  - scheme\_type, 54
  - scope, 55
  - url\_path, 55
  - url\_specific\_part, 55
  - user, 54
- host
  - globus\_url\_t, 54
- Mutual Exclusion, 36
- One-time execution, 47
- password
  - globus\_url\_t, 54
- port
  - globus\_url\_t, 54
- scheme
  - globus\_url\_t, 54
- scheme\_type
  - globus\_url\_t, 54
- scope
  - globus\_url\_t, 55
- Thread-Specific Storage, 45
- Threading, 31
- URL String Parser, 48
- url\_path
  - globus\_url\_t, 55
- url\_specific\_part
  - globus\_url\_t, 55
- user
  - globus\_url\_t, 54