NAME

libcurl − client-side URL transfers

DESCRIPTION

This is a short overview on how to use libcurl in your C programs. There are specific man pages for each function mentioned in here. There are also the *libcurl-easy(3)* man page, the *libcurl-multi(3)* man page, the *libcurl-share(3)* man page and the *libcurl-tutorial(3)* man page for in-depth understanding on how to program with libcurl.

There are many bindings available that bring libcurl access to your favourite language. Look elsewhere for documentation on those.

libcurl has a global constant environment that you must set up and maintain while using libcurl. This essentially means you call *curl_global_init(3)* at the start of your program and *curl_global_cleanup(3)* at the end. See **GLOBAL CONSTANTS** below for details.

To transfer files, you create an "easy handle" using *curl_easy_init(3)* for a single individual transfer (in either direction). You then set your desired set of options in that handle with *curk_easy_setopt(3)*. Options you set with *curl_easy_setopt(3)* stick. They will be used on every repeated use of this handle until you either change the option, or you reset them all with *curl_easy_reset(3)*.

To actually transfer data you have the option of using the "easy" interface, or the "multi" interface.

The easy interface is a synchronous interface with which you call *curl_easy_perform(3)* and let it perform the transfer. When it is completed, the function returns and you can continue. More details are found in the *libcurl-easy(3)* man page.

The multi interface on the other hand is an asynchronous interface, that you call and that performs only a little piece of the transfer on each invoke. It is perfect if you want to do things while the transfer is in progress, or similar. The multi interface allows you to select() on libcurl action, and even to easily download multiple files simultaneously using a single thread. See further details in the *libcurl-multi(3)* man page.

You can have multiple easy handles share certain data, even if they are used in different threads. This magic is setup using the share interface, as described in the *libcurl-share(3)* man page.

There is also a series of other helpful functions to use, including these:

curl_version_info()
: gets detailed libcurl (and other used libraries) version info

curl_getdate()
: converts a date string to time_t

curl_easy_getinfo()
: get information about a performed transfer

curl_formadd()
: helps building an HTTP form POST

curl_formfree()
: free a list built with *curl_formadd(3)*

curl_slist_append()
: builds a linked list

curl_slist_free_all()
: frees a whole curl_slist

LINKING WITH LIBCURL

On unix-like machines, there's a tool named curl-config that gets installed with the rest of the curl stuff when 'make install' is performed.

curl-config is added to make it easier for applications to link with libcurl and developers to learn about libcurl and how to use it.

Run 'curl-config --libs' to get the (additional) linker options you need to link with the particular version of libcurl you've installed. See the *curl-config(1)* man page for further details.

Unix-like operating system that ship libcurl as part of their distributions often don't provide the curl-config tool, but simply install the library and headers in the common path for this purpose.

Many Linux and similar sytems use pkg-config to provide build and link options about libraries and libcurl supports that as well.

LIBCURL SYMBOL NAMES

All public functions in the libcurl interface are prefixed with 'curl_' (with a lowercase c). You can find other functions in the library source code, but other prefixes indicate that the functions are private and may change without further notice in the next release.

Only use documented functions and functionality!

PORTABILITY

libcurl works **exactly** the same, on any of the platforms it compiles and builds on.

THREADS

Never ever call curl-functions simultaneously using the same handle from several threads. libcurl is thread-safe and can be used in any number of threads, but you must use separate curl handles if you want to use libcurl in more than one thread simultaneously.

The global environment functions are not thread-safe. See **GLOBAL CONSTANTS** below for details.

PERSISTENT CONNECTIONS

Persistent connections means that libcurl can re-use the same connection for several transfers, if the conditions are right.

libcurl will **always** attempt to use persistent connections. Whenever you use *curl_easy_perform(3)* or *curl_multi_perform(3)* etc, libcurl will attempt to use an existing connection to do the transfer, and if none exists it'll open a new one that will be subject for re-use on a possible following call to *curl_easy_perform(3)* or *curl_multi_perform(3)*.

To allow libcurl to take full advantage of persistent connections, you should do as many of your file transfers as possible using the same handle.

If you use the easy interface, and you call *curl_easy_cleanup(3)*, all the possibly open connections held by libcurl will be closed and forgotten.

When you've created a multi handle and are using the multi interface, the connection pool is instead kept in the multi handle so closing and creating new easy handles to do transfers will not affect them. Instead all added easy handles can take advantage of the single shared pool.

GLOBAL CONSTANTS

There are a variety of constants that libcurl uses, mainly through its internal use of other libraries, which are too complicated for the library loader to set up. Therefore, a program must call a library function after the program is loaded and running to finish setting up the library code. For example, when libcurl is built for

SSL capability via the GNU TLS library, there is an elaborate tree inside that library that describes the SSL protocol.

curl_global_init() is the function that you must call. This may allocate resources (e.g. the memory for the GNU TLS tree mentioned above), so the companion function *curl_global_cleanup()* releases them.

The basic rule for constructing a program that uses libcurl is this: Call *curl_global_init()*, with a *CURL_GLOBAL_ALL* argument, immediately after the program starts, while it is still only one thread and before it uses libcurl at all. Call *curl_global_cleanup()* immediately before the program exits, when the program is again only one thread and after its last use of libcurl.

You can call both of these multiple times, as long as all calls meet these requirements and the number of calls to each is the same.

It isn't actually required that the functions be called at the beginning and end of the program -- that's just usually the easiest way to do it. It *is* required that the functions be called when no other thread in the program is running.

These global constant functions are *not thread safe*, so you must not call them when any other thread in the program is running. It isn't good enough that no other thread is using libcurl at the time, because these functions internally call similar functions of other libraries, and those functions are similarly thread-unsafe. You can't generally know what these libraries are, or whether other threads are using them.

The global constant situation merits special consideration when the code you are writing to use libcurl is not the main program, but rather a modular piece of a program, e.g. another library. As a module, your code doesn't know about other parts of the program -- it doesn't know whether they use libcurl or not. And its code doesn't necessarily run at the start and end of the whole program.

A module like this must have global constant functions of its own, just like *curl_global_init()* and *curl_global_cleanup()*. The module thus has control at the beginning and end of the program and has a place to call the libcurl functions. Note that if multiple modules in the program use libcurl, they all will separately call the libcurl functions, and that's OK because only the first *curl_global_init()* and the last *curl_global_cleanup()* in a program change anything. (libcurl uses a reference count in static memory).

In a C++ module, it is common to deal with the global constant situation by defining a special class that represents the global constant environment of the module. A program always has exactly one object of the class, in static storage. That way, the program automatically calls the constructor of the object as the program starts up and the destructor as it terminates. As the author of this libcurl-using module, you can make the constructor call *curl_global_init()* and the destructor call *curl_global_cleanup()* and satisfy libcurl's requirements without your user having to think about it.

curl_global_init() has an argument that tells what particular parts of the global constant environment to set up. In order to successfully use any value except *CURL_GLOBAL_ALL* (which says to set up the whole thing), you must have specific knowledge of internal workings of libcurl and all other parts of the program of which it is part.

A special part of the global constant environment is the identity of the memory allocator. *curl_global_init()* selects the system default memory allocator, but you can use *curl_global_init_mem()* to supply one of your own. However, there is no way to use *curl_global_init_mem()* in a modular program -- all modules in the program that might use libcurl would have to agree on one allocator.

There is a failsafe in libcurl that makes it usable in simple situations without you having to worry about the global constant environment at all: *curl_easy_init()* sets up the environment itself if it hasn't been done yet. The resources it acquires to do so get released by the operating system automatically when the program

exits.

This failsafe feature exists mainly for backward compatibility because there was a time when the global functions didn’t exist. Because it is sufficient only in the simplest of programs, it is not recommended for any program to rely on it.