libcurl programming tutorial

NAME

libcurl-tutorial - libcurl programming tutorial

Objective

This document attempts to describe the general principles and some basic approaches to consider when programming with libcurl. The text will focus mainly on the C interface but might apply fairly well on other interfaces as well as they usually follow the C one pretty closely.

This document will refer to 'the user' as the person writing the source code that uses libcurl. That would probably be you or someone in your position. What will be generally referred to as 'the program' will be the collected source code that you write that is using libcurl for transfers. The program is outside libcurl and libcurl is outside of the program.

To get more details on all options and functions described herein, please refer to their respective man pages.

Building

There are many different ways to build C programs. This chapter will assume a Unix style build process. If you use a different build system, you can still read this to get general information that may apply to your environment as well.

Compiling the Program

Your compiler needs to know where the libcurl headers are located. Therefore you must set your compiler's include path to point to the directory where you installed them. The 'curl-config'[3] tool can be used to get this information:

```
$ curl-config --cflags
```

Linking the Program with libcurl

When having compiled the program, you need to link your object files to create a single executable. For that to succeed, you need to link with libcurl and possibly also with other libraries that libcurl itself depends on. Like the OpenSSL libraries, but even some standard OS libraries may be needed on the command line. To figure out which flags to use, once again the 'curl-config' tool comes to the rescue:

$ curl-config --libs

SSL or Not

libcurl can be built and customized in many ways. One of the things that varies from different libraries and builds is the support for SSL-based transfers, like HTTPS and FTPS. If a supported SSL library was detected properly at build-time, libcurl will be built with SSL support. To figure out if an installed libcurl has been built with SSL support enabled, use 'curl-config' like this:

$ curl-config --feature

And if SSL is supported, the keyword 'SSL' will be written to stdout, possibly together with a few other features that could be either on or off on for different libcurls.

See also the "Features libcurl Provides" further down.

autoconf macro

When you write your configure script to detect libcurl and setup variables accordingly, we offer a prewritten macro that probably does everything you need in this area. See docs/libcurl/libcurl.m4 file - it includes docs on how to use it.

Portable Code in a Portable World

The people behind libcurl have put a considerable effort to make libcurl work on a large amount of different operating systems and environments.

You program libcurl the same way on all platforms that libcurl runs on. There are only very few minor considerations that differ. If you just make sure to write your code portable enough, you may very well create yourself a very portable program. libcurl shouldn't stop you from that.

Global Preparation

The program must initialize some of the libcurl functionality globally. That means it should be done exactly once, no matter how many times you intend to use the library. Once for your program's entire life time. This is done using

curl_global_init()

and it takes one parameter which is a bit pattern that tells libcurl what to initialize. Using *CURL_GLOBAL_ALL* will make it initialize all known internal sub modules, and might be a good default option. The current two bits that are specified are:

CURL_GLOBAL_WIN32

which only does anything on Windows machines. When used on a Windows machine, it'll make libcurl initialize the win32 socket stuff. Without having that initialized properly, your program cannot use sockets properly. You should only do this once for each application, so if your program already does this or of another library in use does it, you should not tell libcurl to do this as well.

CURL_GLOBAL_SSL

which only does anything on libcurls compiled and built SSL-enabled. On these systems, this will make libcurl initialize the SSL library properly for this application. This only needs to be done once for each application so if your program or another library already does this, this bit should not be needed.

libcurl has a default protection mechanism that detects if curl_global_init hasn't been called by the time curl_easy_perform is called and if that is the case, libcurl runs the function itself with a guessed bit pattern. Please note that depending solely on this is not considered nice nor very good.

When the program no longer uses libcurl, it should call curl_global_cleanup, which is the opposite of the init call. It will then do the reversed operations to cleanup the resources the curl_global_init call initialized.

Repeated calls to curl_global_init and curl_global_cleanup should be avoided. They should only be called once each.

Features libcurl Provides

It is considered best-practice to determine libcurl features at run-time rather than at build-time (if possible of course). By calling curl_version_info and checking out the details of the returned struct, your program can figure out exactly what the currently running libcurl supports.

Two Interfaces

libcurl first introduced the so called easy interface. All operations in the easy interface are prefixed with 'curl_easy'. The easy interface lets you do single transfers with a synchronous and blocking function call.

libcurl also offers another interface that allows multiple simultaneous transfers in a single thread, the so called multi interface. More about that interface is detailed in a separate chapter further down. You still need to understand the easy interface first, so please continue reading for better understanding.

Handle the Easy libcurl

To use the easy interface, you must first create yourself an easy handle. You need one handle for each easy session you want to perform. Basically, you should use one handle for every thread you plan to use for transferring. You must never share the same handle in multiple threads.

Get an easy handle with

```
 easyhandle = curl_easy_init();
```

It returns an easy handle. Using that you proceed to the next step: setting up your preferred actions. A handle is just a logic entity for the upcoming transfer or series of transfers.

You set properties and options for this handle using curl_easy_setopt. They control how the subsequent transfer or transfers will be made. Options remain set in the handle until set again to something different. They are sticky. Multiple requests using the same handle will use the same options.

If you at any point would like to blank all previously set options for a single easy handle, you can call curl_easy_reset and you can also make a clone of an easy handle (with all its set options) using curl_easy_duphandle.

Many of the options you set in libcurl are "strings", pointers to data terminated with a zero byte. When you set strings with curl_easy_setopt, libcurl makes its own copy so that they don't need to be kept around in your application after being set[4].

One of the most basic properties to set in the handle is the URL. You set your preferred URL to transfer with CURLOPT_URL in a manner similar to:

```
 curl_easy_setopt(handle, CURLOPT_URL, "http://domain.com/");
```

Let's assume for a while that you want to receive data as the URL identifies a remote resource you want to get here. Since you write a sort of application that needs this transfer, I assume that you would like to get the data passed to you directly instead of simply getting it passed to stdout. So, you write your own function that matches this prototype:

```
 size_t write_data(void *buffer, size_t size, size_t nmemb, void *userp);
```

You tell libcurl to pass all data to this function by issuing a function similar to this:

```
 curl_easy_setopt(easyhandle, CURLOPT_WRITEFUNCTION, write_data);
```

You can control what data your callback function gets in the fourth argument by setting another property:

```
 curl_easy_setopt(easyhandle, CURLOPT_WRITEDATA, &internal_struct);
```

Using that property, you can easily pass local data between your application and the function that gets invoked by libcurl. libcurl itself won't touch the data you pass with CURLOPT_WRITEDATA.

libcurl offers its own default internal callback that will take care of the data if you don't set the callback with CURLOPT_WRITEFUNCTION. It will then simply output the received data to stdout. You can have the default callback write the data to a different file handle by passing a 'FILE *' to a file opened for writing with the CURLOPT_WRITEDATA option.

Now, we need to take a step back and have a deep breath. Here's one of those rare platform-dependent nitpicks. Did you spot it? On some platforms[2], libcurl won't be able to operate on files opened by the program. Thus, if you use the default callback and pass in an open file with CURLOPT_WRITEDATA, it will crash. You should therefore avoid this to make your program run fine virtually everywhere.

(CURLOPT_WRITEDATA was formerly known as *CURLOPT_FILE*. Both names still work and do the same thing).

If you're using libcurl as a win32 DLL, you MUST use the CURLOPT_WRITEFUNCTION if you set CURLOPT_WRITEDATA - or you will experience crashes.

There are of course many more options you can set, and we'll get back to a few of them later. Let's instead continue to the actual transfer:

```
 success = curl_easy_perform(easyhandle);
```

curl_easy_perform will connect to the remote site, do the necessary commands and receive the transfer. Whenever it receives data, it calls the callback function we previously set. The function may get one byte at a time, or it may get many kilobytes at once. libcurl delivers as much as possible as often as possible. Your callback function should return the number of bytes it "took care of". If that is not the exact same amount of bytes that was passed to it, libcurl will abort the operation and return with an error code.

When the transfer is complete, the function returns a return code that informs you if it succeeded in its mission or not. If a return code isn't enough for you, you can use the CURLOPT_ERRORBUFFER to point libcurl to a buffer of yours where it'll store a human readable error message as well.

If you then want to transfer another file, the handle is ready to be used again. Mind you, it is even preferred that you re-use an existing handle if you intend to make another transfer. libcurl will then attempt to re-use the previous connection.

For some protocols, downloading a file can involve a complicated process of logging in, setting the transfer mode, changing the current directory and finally transferring the file data. libcurl takes care of all that complication for you. Given simply the URL to a file, libcurl will take care of all the details needed to get the file moved from one machine to another.

Multi-threading Issues

libcurl is thread safe but there are a few exceptions. Refer to libcurl-thread for more information.

When It Doesn't Work

There will always be times when the transfer fails for some reason. You might have set the wrong libcurl option or misunderstood what the libcurl option actually does, or the remote server might return non-standard replies that confuse the library which then confuses your program.

There's one golden rule when these things occur: set the CURLOPT_VERBOSE option to 1. It'll cause the library to spew out the entire protocol details it sends, some internal info and some received protocol data as well (especially when using FTP). If you're using HTTP, adding the headers in the received output to study is also a clever way to get a better understanding why the server behaves the way it does. Include headers in the normal body output with CURLOPT_HEADER set 1.

Of course, there are bugs left. We need to know about them to be able to fix them, so we're quite dependent on your bug reports! When you do report suspected bugs in libcurl, please include as many details as you possibly can: a protocol dump that CURLOPT_VERBOSE produces, library version, as much as possible of your code that uses libcurl, operating system name and version, compiler name and version etc.

If CURLOPT_VERBOSE is not enough, you increase the level of debug data your application receive by using the CURLOPT_DEBUGFUNCTION.

Getting some in-depth knowledge about the protocols involved is never wrong, and if you're trying to do funny things, you might very well understand libcurl and how to use it better if you study the appropriate RFC documents at least briefly.

Upload Data to a Remote Site

libcurl tries to keep a protocol independent approach to most transfers, thus uploading to a remote FTP site is very similar to uploading data to an HTTP server with a PUT request.

Of course, first you either create an easy handle or you re-use one existing one. Then you set the URL to operate on just like before. This is the remote URL, that we now will upload.

Since we write an application, we most likely want libcurl to get the upload data by asking us for it. To make it do that, we set the read callback and the custom pointer libcurl will pass to our read callback. The read callback should have a prototype similar to:

```
size_t function(char *bufptr, size_t size, size_t nitems, void *userp);
```

Where bufptr is the pointer to a buffer we fill in with data to upload and size*nitems is the size of the buffer and therefore also the maximum amount of data we can return to libcurl in this call. The 'userp' pointer is the custom pointer we set to point to a struct of ours to pass private data between the application and the callback.

```
curl_easy_setopt(easyhandle, CURLOPT_READFUNCTION, read_function);
```

```
curl_easy_setopt(easyhandle, CURLOPT_READDATA, &filedata);
```

Tell libcurl that we want to upload:

```
curl_easy_setopt(easyhandle, CURLOPT_UPLOAD, 1L);
```

A few protocols won't behave properly when uploads are done without any prior knowledge of the expected file size. So, set the upload file size using the CURLOPT_INFILESIZE_LARGE for all known file sizes like this[1]:

```
/* in this example, file_size must be an curl_off_t variable */
curl_easy_setopt(easyhandle, CURLOPT_INFILESIZE_LARGE, file_size);
```

When you call curl_easy_perform this time, it'll perform all the necessary operations and when it has invoked the upload it'll call your supplied callback to get the data to upload. The program should return as much data as possible in every invoke, as that is likely to make the upload perform as fast as possible. The callback should return the number of bytes it wrote in the buffer. Returning 0 will signal the end of the upload.

Passwords

Many protocols use or even require that user name and password are provided to be able to download or upload the data of your choice. libcurl offers several ways to specify them.

Most protocols support that you specify the name and password in the URL itself. libcurl will detect this and use them accordingly. This is written like this:

```
protocol://user:password@example.com/path/
```

If you need any odd letters in your user name or password, you should enter them URL encoded, as %XX where XX is a two-digit hexadecimal number.

libcurl also provides options to set various passwords. The user name and password as shown embedded in the URL can instead get set with the CURLOPT_USERPWD option. The argument passed to libcurl should be a char * to a string in the format "user:password". In a manner like this:

```
 curl_easy_setopt(easyhandle, CURLOPT_USERPWD, "myname:thesecret");
```

Another case where name and password might be needed at times, is for those users who need to authenticate themselves to a proxy they use. libcurl offers another option for this, the CURLOPT_PROXYUSERPWD. It is used quite similar to the CURLOPT_USERPWD option like this:

```
 curl_easy_setopt(easyhandle, CURLOPT_PROXYUSERPWD, "myname:thesecret");
```

There's a long time Unix "standard" way of storing FTP user names and passwords, namely in the $HOME/.netrc file. The file should be made private so that only the user may read it (see also the "Security Considerations" chapter), as it might contain the password in plain text. libcurl has the ability to use this file to figure out what set of user name and password to use for a particular host. As an extension to the normal functionality, libcurl also supports this file for non-FTP protocols such as HTTP. To make curl use this file, use the CURLOPT_NETRC option:

```
 curl_easy_setopt(easyhandle, CURLOPT_NETRC, 1L);
```

And a very basic example of how such a .netrc file may look like:

```
 machine myhost.mydomain.com
 login userlogin
 password secretword
```

All these examples have been cases where the password has been optional, or at least you could leave it out and have libcurl attempt to do its job without it. There are times when the password isn't optional, like when you're using an SSL private key for secure transfers.

To pass the known private key password to libcurl:

```
 curl_easy_setopt(easyhandle, CURLOPT_KEYPASSWD, "keypassword");
```

HTTP Authentication

The previous chapter showed how to set user name and password for getting URLs that require authentication. When using the HTTP protocol, there are many different ways a client can provide those credentials to the server and you can control which way libcurl will (attempt to) use them. The default HTTP authentication method is called 'Basic', which is sending the name and password in clear-text in the HTTP request, base64-encoded. This is insecure.

At the time of this writing, libcurl can be built to use: Basic, Digest, NTLM, Negotiate (SPNEGO). You can tell libcurl which one to use with CURLOPT_HTTPAUTH as in:

```
 curl_easy_setopt(easyhandle, CURLOPT_HTTPAUTH, CURLAUTH_DIGEST);
```

And when you send authentication to a proxy, you can also set authentication type the same way but instead with CURLOPT_PROXYAUTH:

```
 curl_easy_setopt(easyhandle, CURLOPT_PROXYAUTH, CURLAUTH_NTLM);
```

Both these options allow you to set multiple types (by ORing them together), to make libcurl pick the most secure one out of the types the server/proxy claims to support. This method does however add a round-trip since libcurl must first ask the server what it supports:

```
 curl_easy_setopt(easyhandle, CURLOPT_HTTPAUTH,
 CURLAUTH_DIGEST|CURLAUTH_BASIC);
```

For convenience, you can use the 'CURLAUTH_ANY' define (instead of a list with specific types) which allows libcurl to use whatever method it wants.

When asking for multiple types, libcurl will pick the available one it considers "best" in its own internal order of preference.

HTTP POSTing

We get many questions regarding how to issue HTTP POSTs with libcurl the proper way. This chapter will thus include examples using both different versions of HTTP POST that libcurl supports.

The first version is the simple POST, the most common version, that most HTML pages using the <form> tag uses. We provide a pointer to the data and tell libcurl to post it all to the remote site:

```
    char *data="name=daniel&project=curl";
    curl_easy_setopt(easyhandle, CURLOPT_POSTFIELDS, data);
    curl_easy_setopt(easyhandle, CURLOPT_URL, "http://posthere.com/");

    curl_easy_perform(easyhandle); /* post away! */
```

Simple enough, huh? Since you set the POST options with the CURLOPT_POSTFIELDS, this automatically switches the handle to use POST in the upcoming request.

Ok, so what if you want to post binary data that also requires you to set the Content-Type: header of the post? Well, binary posts prevent libcurl from being able to do strlen() on the data to figure out the size, so therefore we must tell libcurl the size of the post data. Setting headers in libcurl requests are done in a generic way, by building a list of our own headers and then passing that list to libcurl.

```
 struct curl_slist *headers=NULL;
 headers = curl_slist_append(headers, "Content-Type: text/xml");

 /* post binary data */
 curl_easy_setopt(easyhandle, CURLOPT_POSTFIELDS, binaryptr);

 /* set the size of the postfields data */
 curl_easy_setopt(easyhandle, CURLOPT_POSTFIELDSIZE, 23L);

 /* pass our list of custom made headers */
 curl_easy_setopt(easyhandle, CURLOPT_HTTPHEADER, headers);

 curl_easy_perform(easyhandle); /* post away! */

 curl_slist_free_all(headers); /* free the header list */
```

While the simple examples above cover the majority of all cases where HTTP POST operations are required, they don't do multi-part formposts. Multi-part formposts were introduced as a better way to post (possibly large) binary data and were first documented in the RFC 1867 (updated in RFC 2388). They're called multi-part because they're built by a chain of parts, each part being a single unit of data. Each part has its own name and contents. You can in fact create and post a multi-part formpost with the regular libcurl POST support described above, but that would require that you build a formpost yourself and provide to libcurl. To make that easier, libcurl provides a MIME API consisting in several functions: using those, you can create and fill a multi-part form. Function curl_mime_init creates a multi-part body; you can then append new parts to a multi-part body using curl_mime_addpart. There are three possible data sources for a part: memory using curl_mime_data, file using curl_mime_filedata and user-defined data read callback using curl_mime_data_cb. curl_mime_name sets a part's (i.e.: form field) name, while curl_mime_filename fills in the remote file name. With curl_mime_type, you can tell the MIME type of a part, curl_mime_headers allows defining the part's headers. When a multi-part body is no longer needed, you can destroy it using curl_mime_free.

The following example sets two simple text parts with plain textual contents, and then a file with binary contents and uploads the whole thing.

```
curl_mime *multipart = curl_mime_init(easyhandle);
curl_mimepart *part = curl_mime_addpart(mutipart);
curl_mime_name(part, "name");
curl_mime_data(part, "daniel", CURL_ZERO_TERMINATED);
part = curl_mime_addpart(mutipart);
curl_mime_name(part, "project");
curl_mime_data(part, "curl", CURL_ZERO_TERMINATED);
part = curl_mime_addpart(mutipart);
curl_mime_name(part, "logotype-image");
curl_mime_filedata(part, "curl.png");

/* Set the form info */
curl_easy_setopt(easyhandle, CURLOPT_MIMEPOST, multipart);

curl_easy_perform(easyhandle); /* post away! */

/* free the post data again */
curl_mime_free(multipart);
```

To post multiple files for a single form field, you must supply each file in a separate part, all with the same field name. Although function curl_mime_subparts implements nested multi-parts, this way of multiple files posting is deprecated by RFC 7578, chapter 4.3.

To set the data source from an already opened FILE pointer, use:

```
curl_mime_data_cb(part, filesize, (curl_read_callback) fread,
                  (curl_seek_callback) fseek, NULL, filepointer);
```

A deprecated curl_formadd function is still supported in libcurl. It should however not be used anymore for new designs and programs using it ought to be converted to the MIME API. It is however described here as an aid to conversion.

Using *curl_formadd*, you add parts to the form. When you're done adding parts, you post the whole form.

The MIME API example above is expressed as follows using this function:

```
 struct curl_httppost *post=NULL;
 struct curl_httppost *last=NULL;
 curl_formadd(&post, &last,
              CURLFORM_COPYNAME, "name",
              CURLFORM_COPYCONTENTS, "daniel", CURLFORM_END);
 curl_formadd(&post, &last,
              CURLFORM_COPYNAME, "project",
              CURLFORM_COPYCONTENTS, "curl", CURLFORM_END);
 curl_formadd(&post, &last,
              CURLFORM_COPYNAME, "logotype-image",
              CURLFORM_FILECONTENT, "curl.png", CURLFORM_END);

 /* Set the form info */
 curl_easy_setopt(easyhandle, CURLOPT_HTTPPOST, post);

 curl_easy_perform(easyhandle); /* post away! */

 /* free the post data again */
 curl_formfree(post);
```

Multipart formposts are chains of parts using MIME-style separators and headers. It means that each one of these separate parts get a few headers set that describe the individual content-type, size etc. To enable your application to handicraft this formpost even more, libcurl allows you to supply your own set of custom headers to such an individual form part. You can of course supply headers to as many parts as you like, but this little example will show how you set headers to one specific part when you add that to the post handle:

```
 struct curl_slist *headers=NULL;
 headers = curl_slist_append(headers, "Content-Type: text/xml");

 curl_formadd(&post, &last,
              CURLFORM_COPYNAME, "logotype-image",
              CURLFORM_FILECONTENT, "curl.xml",
              CURLFORM_CONTENTHEADER, headers,
              CURLFORM_END);

 curl_easy_perform(easyhandle); /* post away! */

 curl_formfree(post); /* free post */
 curl_slist_free_all(headers); /* free custom header list */
```

Since all options on an easyhandle are "sticky", they remain the same until changed even if you do call curl_easy_perform, you may need to tell curl to go back to a plain GET request if you intend to do one as your next request. You force an easyhandle to go back to GET by using the CURLOPT_HTTPGET option:

```
 curl_easy_setopt(easyhandle, CURLOPT_HTTPGET, 1L);
```

Just setting CURLOPT_POSTFIELDS to "" or NULL will *not* stop libcurl from doing a POST. It will just make it POST without any data to send!

Converting from deprecated form API to MIME API

Four rules have to be respected in building the multi-part:
- The easy handle must be created before building the multi-part.

- The multi-part is always created by a call to curl_mime_init(easyhandle).
- Each part is created by a call to curl_mime_addpart(multipart).
- When complete, the multi-part must be bound to the easy handle using CURLOPT_MIMEPOST instead of CURLOPT_HTTPPOST.

Here are some example of *curl_formadd* calls to MIME API sequences:

```
 curl_formadd(&post, &last,
              CURLFORM_COPYNAME, "id",
              CURLFORM_COPYCONTENTS, "daniel", CURLFORM_END);
              CURLFORM_CONTENTHEADER, headers,
              CURLFORM_END);
```

becomes:

```
 part = curl_mime_addpart(multipart);
 curl_mime_name(part, "id");
 curl_mime_data(part, "daniel", CURL_ZERO_TERMINATED);
 curl_mime_headers(part, headers, FALSE);
```

Setting the last *curl_mime_headers* argument to TRUE would have caused the headers to be automatically released upon destroyed the multi-part, thus saving a clean-up call to curl_slist_free_all.

```
 curl_formadd(&post, &last,
              CURLFORM_PTRNAME, "logotype-image",
              CURLFORM_FILECONTENT, "-",
              CURLFORM_END);
```

becomes:

```
 part = curl_mime_addpart(multipart);
 curl_mime_name(part, "logotype-image");
 curl_mime_data_cb(part, (curl_off_t) -1, fread, fseek, NULL, stdin);
```

curl_mime_name always copies the field name. The special file name "-" is not supported by *curl_mime_file*: to read an open file, use a callback source using fread(). The transfer will be chunked since the data size is unknown.

```
 curl_formadd(&post, &last,
              CURLFORM_COPYNAME, "datafile[]",
              CURLFORM_FILE, "file1",
              CURLFORM_FILE, "file2",
              CURLFORM_END);
```

becomes:

```
 part = curl_mime_addpart(multipart);
 curl_mime_name(part, "datafile[]");
 curl_mime_filedata(part, "file1");
 part = curl_mime_addpart(multipart);
 curl_mime_name(part, "datafile[]");
 curl_mime_filedata(part, "file2");
```

The deprecated multipart/mixed implementation of multiple files field is translated to two distinct parts with the same name.

```
 curl_easy_setopt(easyhandle, CURLOPT_READFUNCTION, myreadfunc);
 curl_formadd(&post, &last,
              CURLFORM_COPYNAME, "stream",
              CURLFORM_STREAM, arg,
              CURLFORM_CONTENTLEN, (curl_off_t) datasize,
              CURLFORM_FILENAME, "archive.zip",
              CURLFORM_CONTENTTYPE, "application/zip",
              CURLFORM_END);
```

becomes:

```
 part = curl_mime_addpart(multipart);
 curl_mime_name(part, "stream");
 curl_mime_data_cb(part, (curl_off_t) datasize,
                   myreadfunc, NULL, NULL, arg);
 curl_mime_filename(part, "archive.zip");
 curl_mime_type(part, "application/zip");
```

CURLOPT_READFUNCTION callback is not used: it is replace by directly setting the part source data from the callback read function.

```
 curl_formadd(&post, &last,
              CURLFORM_COPYNAME, "memfile",
              CURLFORM_BUFFER, "memfile.bin",
              CURLFORM_BUFFERPTR, databuffer,
              CURLFORM_BUFFERLENGTH, (long) sizeof databuffer,
              CURLFORM_END);
```

becomes:

```
 part = curl_mime_addpart(multipart);
 curl_mime_name(part, "memfile");
 curl_mime_data(part, databuffer, (curl_off_t) sizeof databuffer);
 curl_mime_filename(part, "memfile.bin");
```

curl_mime_data always copies the initial data: data buffer is thus free for immediate reuse.

```
 curl_formadd(&post, &last,
              CURLFORM_COPYNAME, "message",
              CURLFORM_FILECONTENT, "msg.txt",
              CURLFORM_END);
```

becomes:

```
 part = curl_mime_addpart(multipart);
 curl_mime_name(part, "message");
 curl_mime_filedata(part, "msg.txt");
 curl_mime_filename(part, NULL);
```

Use of *curl_mime_filedata* sets the remote file name as a side effect: it is therefore necessary to clear it for *CURLFORM_FILECONTENT* emulation.

Showing Progress

For historical and traditional reasons, libcurl has a built-in progress meter that can be switched on and then makes it present a progress meter in your terminal.

Switch on the progress meter by, oddly enough, setting CURLOPT_NOPROGRESS to zero. This option is set to 1 by default.

For most applications however, the built-in progress meter is useless and what instead is interesting is the ability to specify a progress callback. The function pointer you pass to libcurl will then be called on irregular intervals with information about the current transfer.

Set the progress callback by using CURLOPT_PROGRESSFUNCTION. And pass a pointer to a function that matches this prototype:

```
 int progress_callback(void *clientp,
                       double dltotal,
                       double dlnow,
                       double ultotal,
                       double ulnow);
```

If any of the input arguments is unknown, a 0 will be passed. The first argument, the 'clientp' is the pointer you pass to libcurl with CURLOPT_PROGRESSDATA. libcurl won't touch it.

libcurl with C++

There's basically only one thing to keep in mind when using C++ instead of C when interfacing libcurl:

The callbacks CANNOT be non-static class member functions

Example C++ code:

```
class AClass {
    static size_t write_data(void *ptr, size_t size, size_t nmemb,
                             void *ourpointer)
    {
      /* do what you want with the data */
    }
 }
```

Proxies

What "proxy" means according to Merriam-Webster: "a person authorized to act for another" but also "the agency, function, or office of a deputy who acts as a substitute for another".

Proxies are exceedingly common these days. Companies often only offer Internet access to employees through their proxies. Network clients or user-agents ask the proxy for documents, the proxy does the actual request and then it returns them.

libcurl supports SOCKS and HTTP proxies. When a given URL is wanted, libcurl will ask the proxy for it instead of trying to connect to the actual host identified in the URL.

If you're using a SOCKS proxy, you may find that libcurl doesn't quite support all operations through it.

For HTTP proxies: the fact that the proxy is an HTTP proxy puts certain restrictions on what can actually happen. A requested URL that might not be a HTTP URL will be still be passed to the HTTP proxy to deliver back to libcurl. This happens transparently, and an application may not need to know. I say "may", because at times it is very important to understand that all operations over an HTTP proxy use the HTTP protocol. For example, you can't invoke your own custom FTP commands or even proper FTP directory listings.

Proxy Options

To tell libcurl to use a proxy at a given port number:

```
curl_easy_setopt(easyhandle, CURLOPT_PROXY, "proxy-host.com:8080");
```

Some proxies require user authentication before allowing a request, and you pass that information similar to this:

```
curl_easy_setopt(easyhandle, CURLOPT_PROXYUSERPWD, "user:password");
```

If you want to, you can specify the host name only in the CURLOPT_PROXY option, and set the port number separately with CURLOPT_PROXYPORT.

Tell libcurl what kind of proxy it is with CURLOPT_PROXYTYPE (if not, it will default to assume an HTTP proxy):

```
curl_easy_setopt(easyhandle, CURLOPT_PROXYTYPE, CURLPROXY_SOCKS4);
```

Environment Variables

libcurl automatically checks and uses a set of environment variables to know what proxies to use for certain protocols. The names of the variables are following an ancient de facto standard and are built up as "[protocol]_proxy" (note the lower casing). Which makes the variable 'http_proxy' checked for a name of a proxy to use when the input URL is HTTP. Following the same rule, the variable named 'ftp_proxy' is checked for FTP URLs. Again, the proxies are always HTTP proxies, the different names of the variables simply allows different HTTP proxies to be used.

The proxy environment variable contents should be in the format "[protocol://][user:password@]machine[:port]". Where the protocol:// part is simply ignored if present (so http://proxy and bluerk://proxy will do the same) and the optional port number specifies on which port the proxy operates on the host. If not specified, the internal default port number will be used and that is most likely *not* the one you would like it to be.

There are two special environment variables. 'all_proxy' is what sets proxy for any URL in case the protocol specific variable wasn't set, and 'no_proxy' defines a list of hosts that should not use a proxy even though a variable may say so. If 'no_proxy' is a plain asterisk ("*") it matches all hosts.

To explicitly disable libcurl's checking for and using the proxy environment variables, set the proxy name to "" - an empty string - with CURLOPT_PROXY.

SSL and Proxies

SSL is for secure point-to-point connections. This involves strong encryption and similar things, which effectively makes it impossible for a proxy to operate as a "man in between" which the proxy's task is, as previously discussed. Instead, the only way to have SSL work over an HTTP proxy is to ask the proxy to tunnel trough everything without being able to check or fiddle with the traffic.

Opening an SSL connection over an HTTP proxy is therefore a matter of asking the proxy for a straight connection to the target host on a specified port. This is made with the HTTP request CONNECT. ("please mr proxy, connect me to that remote host").

Because of the nature of this operation, where the proxy has no idea what kind of data that is passed in and out through this tunnel, this breaks some of the very few advantages that come from using a proxy, such as caching. Many organizations prevent this kind of tunneling to other destination port numbers than 443 (which is the default HTTPS port number).

Tunneling Through Proxy

As explained above, tunneling is required for SSL to work and often even restricted to the operation intended for SSL; HTTPS.

This is however not the only time proxy-tunneling might offer benefits to you or your application.

As tunneling opens a direct connection from your application to the remote machine, it suddenly also re-introduces the ability to do non-HTTP operations over an HTTP proxy. You can in fact use things such as FTP upload or FTP custom commands this way.

Again, this is often prevented by the administrators of proxies and is rarely allowed.

Tell libcurl to use proxy tunneling like this:

```
 curl_easy_setopt(easyhandle, CURLOPT_HTTPPROXYTUNNEL, 1L);
```

In fact, there might even be times when you want to do plain HTTP operations using a tunnel like this, as it then enables you to operate on the remote server instead of asking the proxy to do so. libcurl will not stand in the way for such innovative actions either!

Proxy Auto-Config

Netscape first came up with this. It is basically a web page (usually using a .pac extension) with a Javascript that when executed by the browser with the requested URL as input, returns information to the browser on how to connect to the URL. The returned information might be "DIRECT" (which means no proxy should be used), "PROXY host:port" (to tell the browser where the proxy for this particular URL is) or "SOCKS host:port" (to direct the browser to a SOCKS proxy).

libcurl has no means to interpret or evaluate Javascript and thus it doesn't support this. If you get yourself in a position where you face this nasty invention, the following advice have been mentioned and used in the past:

- Depending on the Javascript complexity, write up a script that translates it to another language and execute that.

- Read the Javascript code and rewrite the same logic in another language.

- Implement a Javascript interpreter; people have successfully used the Mozilla Javascript engine in the past.

- Ask your admins to stop this, for a static proxy setup or similar.

Persistence Is The Way to Happiness

Re-cycling the same easy handle several times when doing multiple requests is the way to go.

After each single curl_easy_perform operation, libcurl will keep the connection alive and open. A subsequent request using the same easy handle to the same host might just be able to use the already open connection! This reduces network impact a lot.

Even if the connection is dropped, all connections involving SSL to the same host again, will benefit from libcurl's session ID cache that drastically reduces re-connection time.

FTP connections that are kept alive save a lot of time, as the command- response round-trips are skipped, and also you don't risk getting blocked without permission to login again like on many FTP servers only allowing N persons to be logged in at the same time.

libcurl caches DNS name resolving results, to make lookups of a previously looked up name a lot faster.

Other interesting details that improve performance for subsequent requests may also be added in the future.

Each easy handle will attempt to keep the last few connections alive for a while in case they are to be used again. You can set the size of this "cache" with the CURLOPT_MAXCONNECTS option. Default is 5. There is very seldom any point in changing this value, and if you think of changing this it is often just a matter of thinking again.

To force your upcoming request to not use an already existing connection (it will even close one first if there happens to be one alive to the same host you're about to operate on), you can do that by setting CURLOPT_FRESH_CONNECT to 1. In a similar spirit, you can also forbid the upcoming request to be "lying" around and possibly get re-used after the request by setting CURLOPT_FORBID_REUSE to 1.

HTTP Headers Used by libcurl

When you use libcurl to do HTTP requests, it'll pass along a series of headers automatically. It might be good for you to know and understand these. You can replace or remove them by using the CURLOPT_HTTPHEADER option.

`Host`

This header is required by HTTP 1.1 and even many 1.0 servers and should be the name of the server we want to talk to. This includes the port number if anything but default.

`Accept`

"*/*".

`Expect`

When doing POST requests, libcurl sets this header to "100-continue" to ask the server for an "OK" message before it proceeds with sending the data part of the post. If the POSTed data amount is deemed "small", libcurl will not use this header.

Customizing Operations

There is an ongoing development today where more and more protocols are built upon HTTP for transport. This has obvious benefits as HTTP is a tested and reliable protocol that is widely deployed and has excellent proxy-support.

When you use one of these protocols, and even when doing other kinds of programming you may need to change the traditional HTTP (or FTP or...) manners. You may need to change words, headers or various data.

libcurl is your friend here too.

CUSTOMREQUEST

If just changing the actual HTTP request keyword is what you want, like when GET, HEAD or POST is not good enough for you, CURLOPT_CUSTOMREQUEST is there for you. It is very simple to use:

```
curl_easy_setopt(easyhandle, CURLOPT_CUSTOMREQUEST, "MYOWNREQUEST");
```

When using the custom request, you change the request keyword of the actual request you are performing. Thus, by default you make a GET request but you can also make a POST operation (as described before) and then replace the POST keyword if you want to. You're the boss.

Modify Headers

HTTP-like protocols pass a series of headers to the server when doing the request, and you're free to pass any amount of extra headers that you think fit. Adding headers is this easy:

```
struct curl_slist *headers=NULL; /* init to NULL is important */

headers = curl_slist_append(headers, "Hey-server-hey: how are you?");
headers = curl_slist_append(headers, "X-silly-content: yes");

/* pass our list of custom made headers */
curl_easy_setopt(easyhandle, CURLOPT_HTTPHEADER, headers);

curl_easy_perform(easyhandle); /* transfer http */

curl_slist_free_all(headers); /* free the header list */
```

... and if you think some of the internally generated headers, such as Accept: or Host: don't contain the data you want them to contain, you can replace them by simply setting them too:

```
headers = curl_slist_append(headers, "Accept: Agent-007");
headers = curl_slist_append(headers, "Host: munged.host.line");
```

Delete Headers

If you replace an existing header with one with no contents, you will prevent the header from being sent. For instance, if you want to completely prevent the "Accept:" header from being sent, you can disable it with code similar to this:

```
headers = curl_slist_append(headers, "Accept:");
```

Both replacing and canceling internal headers should be done with careful consideration and you should be aware that you may violate the HTTP protocol when doing so.

Enforcing chunked transfer-encoding

By making sure a request uses the custom header "Transfer-Encoding: chunked" when doing a non-GET HTTP operation, libcurl will switch over to "chunked" upload, even though the size of the data to upload might be known. By default, libcurl usually switches over to chunked upload automatically if the upload data size is unknown.

HTTP Version

All HTTP requests includes the version number to tell the server which version we support. libcurl speaks HTTP 1.1 by default. Some very old servers don't like getting 1.1-requests and when dealing with stubborn old things like that, you can tell libcurl to use 1.0 instead by doing something like this:

curl_easy_setopt(easyhandle, CURLOPT_HTTP_VERSION, CURL_HTTP_VERSION_1_0);

FTP Custom Commands

Not all protocols are HTTP-like, and thus the above may not help you when you want to make, for example, your FTP transfers to behave differently.

Sending custom commands to an FTP server means that you need to send the commands exactly as the FTP server expects them (RFC959 is a good guide here), and you can only use commands that work on the control-connection alone. All kinds of commands that require data interchange and thus need a data-connection must be left to libcurl's own judgement. Also be aware that libcurl will do its very best to change directory to the target directory before doing any transfer, so if you change directory (with CWD or similar) you might confuse libcurl and then it might not attempt to transfer the file in the correct remote directory.

A little example that deletes a given file before an operation:

```
headers = curl_slist_append(headers, "DELE file-to-remove");

/* pass the list of custom commands to the handle */
curl_easy_setopt(easyhandle, CURLOPT_QUOTE, headers);

curl_easy_perform(easyhandle); /* transfer ftp data! */

curl_slist_free_all(headers); /* free the header list */
```

If you would instead want this operation (or chain of operations) to happen _after_ the data transfer took place the option to curl_easy_setopt would instead be called CURLOPT_POSTQUOTE and used the exact same way.

The custom FTP command will be issued to the server in the same order they are added to the list, and if a command gets an error code returned back from the server, no more commands will be issued and libcurl will bail out with an error code (CURLE_QUOTE_ERROR). Note that if you use CURLOPT_QUOTE to send commands before a transfer, no transfer will actually take place when a quote command has failed.

If you set the CURLOPT_HEADER to 1, you will tell libcurl to get information about the target file and output "headers" about it. The headers will be in "HTTP-style", looking like they do in HTTP.

The option to enable headers or to run custom FTP commands may be useful to combine with CURLOPT_NOBODY. If this option is set, no actual file content transfer will be performed.

FTP Custom CUSTOMREQUEST

If you do want to list the contents of an FTP directory using your own defined FTP command, CURLOPT_CUSTOMREQUEST will do just that. "NLST" is the default one for listing directories but you're free to pass in your idea of a good alternative.

Cookies Without Chocolate Chips

In the HTTP sense, a cookie is a name with an associated value. A server sends the name and value to the client, and expects it to get sent back on every subsequent request to the server that matches the particular conditions set. The conditions include that the domain name and path match and that the cookie hasn't become too old.

In real-world cases, servers send new cookies to replace existing ones to update them. Server use cookies to "track" users and to keep "sessions".

Cookies are sent from server to clients with the header Set-Cookie: and they're sent from clients to servers with the Cookie: header.

To just send whatever cookie you want to a server, you can use CURLOPT_COOKIE to set a cookie string like this:

```
 curl_easy_setopt(easyhandle, CURLOPT_COOKIE, "name1=var1; name2=var2;");
```

In many cases, that is not enough. You might want to dynamically save whatever cookies the remote server passes to you, and make sure those cookies are then used accordingly on later requests.

One way to do this, is to save all headers you receive in a plain file and when you make a request, you tell libcurl to read the previous headers to figure out which cookies to use. Set the header file to read cookies from with CURLOPT_COOKIEFILE.

The CURLOPT_COOKIEFILE option also automatically enables the cookie parser in libcurl. Until the cookie parser is enabled, libcurl will not parse or understand incoming cookies and they will just be ignored. However, when the parser is enabled the cookies will be understood and the cookies will be kept in memory and used properly in subsequent requests when the same handle is used. Many times this is enough, and you may not have to save the cookies to disk at all. Note that the file you specify to CURLOPT_COOKIEFILE doesn't have to exist to enable the parser, so a common way to just enable the parser and not read any cookies is to use the name of a file you know doesn't exist.

If you would rather use existing cookies that you've previously received with your Netscape or Mozilla browsers, you can make libcurl use that cookie file as input. The CURLOPT_COOKIEFILE is used for that too, as libcurl will automatically find out what kind of file it is and act accordingly.

Perhaps the most advanced cookie operation libcurl offers, is saving the entire internal cookie state back into a Netscape/Mozilla formatted cookie file. We call that the cookiejar. When you set a file name with CURLOPT_COOKIEJAR, that file name will be created and all received cookies will be stored in it when curl_easy_cleanup is called. This enables cookies to get passed on properly between multiple handles without any information getting lost.

FTP Peculiarities We Need

FTP transfers use a second TCP/IP connection for the data transfer. This is usually a fact you can forget and ignore but at times this fact will come back to haunt you. libcurl offers several different ways to customize how the second connection is being made.

libcurl can either connect to the server a second time or tell the server to connect back to it. The first option is the default and it is also what works best for all the people behind firewalls, NATs or IP-masquerading setups. libcurl then tells the server to open up a new port and wait for a second connection. This is by default attempted with EPSV first, and if that doesn't work it tries PASV instead. (EPSV is an extension to the original FTP spec and does not exist nor work on all FTP servers.)

You can prevent libcurl from first trying the EPSV command by setting CURLOPT_FTP_USE_EPSV to zero.

In some cases, you will prefer to have the server connect back to you for the second connection. This might be when the server is perhaps behind a firewall or something and only allows connections on a single port. libcurl then informs the remote server which IP address and port number to connect to. This is made with the CURLOPT_FTPPORT option. If you set it to "-", libcurl will use your system's "default IP address". If you want to use a particular IP, you can set the full IP address, a host name to resolve to an IP address or even a local network interface name that libcurl will get the IP address from.

When doing the "PORT" approach, libcurl will attempt to use the EPRT and the LPRT before trying PORT, as they work with more protocols. You can disable this behavior by setting CURLOPT_FTP_USE_EPRT to zero.

MIME API revisited for SMTP and IMAP

In addition to support HTTP multi-part form fields, the MIME API can be used to build structured e-mail messages and send them via SMTP or append such messages to IMAP directories.

A structured e-mail message may contain several parts: some are displayed inline by the MUA, some are attachments. Parts can also be structured as multi-part, for example to include another e-mail message or to offer several text formats alternatives. This can be nested to any level.

To build such a message, you prepare the nth-level multi-part and then include it as a source to the parent multi-part using function curl_mime_subparts. Once it has been bound to its parent multi-part, a nth-level multi-part belongs to it and should not be freed explicitly.

E-mail messages data is not supposed to be non-ascii and line length is limited: fortunately, some transfer encodings are defined by the standards to support the transmission of such incompatible data. Function curl_mime_encoder tells a part that its source data must be encoded before being sent. It also generates the corresponding header for that part. If the part data you want to send is already encoded in such a scheme, do not use this function (this would over-encode it), but explicitly set the corresponding part header.

Upon sending such a message, libcurl prepends it with the header list set with CURLOPT_HTTPHEADER, as 0th-level mime part headers.

Here is an example building an e-mail message with an inline plain/html text alternative and a file attachment encoded in base64:

```
curl_mime *message = curl_mime_init(easyhandle);

/* The inline part is an alternative proposing the html and the text
   versions of the e-mail. */
curl_mime *alt = curl_mime_init(easyhandle);

/* HTML message. */
curl_mimepart *part = curl_mime_addpart(alt);
curl_mime_data(part, "<html><body><p>This is HTML</p></body></html>",
                     CURL_ZERO_TERMINATED);
curl_mime_type(part, "text/html");

/* Text message. */
part = curl_mime_addpart(alt);
curl_mime_data(part, "This is plain text message",
                     CURL_ZERO_TERMINATED);

/* Create the inline part. */
part = curl_mime_addpart(message);
curl_mime_subparts(part, alt);
curl_mime_type(part, "multipart/alternative");
struct curl_slist *headers = curl_slist_append(NULL,
                  "Content-Disposition: inline");
curl_mime_headers(part, headers, TRUE);

/* Add the attachment. */
part = curl_mime_addpart(message);
curl_mime_filedata(part, "manual.pdf");
curl_mime_encoder(part, "base64");

/* Build the mail headers. */
headers = curl_slist_append(NULL, "From: me@example.com");
headers = curl_slist_append(headers, "To: you@example.com");

/* Set these into the easy handle. */
curl_easy_setopt(easyhandle, CURLOPT_HTTPHEADER, headers);
curl_easy_setopt(easyhandle, CURLOPT_MIMEPOST, mime);
```

It should be noted that appending a message to an IMAP directory requires the message size to be known prior upload. It is therefore not possible to include parts with unknown data size in this context.

Headers Equal Fun

Some protocols provide "headers", meta-data separated from the normal data. These headers are by default not included in the normal data stream, but you can make them appear in the data stream by setting CURLOPT_HEADER to 1.

What might be even more useful, is libcurl's ability to separate the headers from the data and thus make the callbacks differ. You can for example set a different pointer to pass to the ordinary write callback by setting CURLOPT_HEADERDATA.

Or, you can set an entirely separate function to receive the headers, by using CURLOPT_HEADERFUNCTION.

The headers are passed to the callback function one by one, and you can depend on that fact. It makes it easier for you to add custom header parsers etc.

"Headers" for FTP transfers equal all the FTP server responses. They aren't actually true headers, but in this case we pretend they are! ;-)

Post Transfer Information

See curl_easy_getinfo.

The multi Interface

The easy interface as described in detail in this document is a synchronous interface that transfers one file at a time and doesn't return until it is done.

The multi interface, on the other hand, allows your program to transfer multiple files in both directions at the same time, without forcing you to use multiple threads. The name might make it seem that the multi interface is for multi-threaded programs, but the truth is almost the reverse. The multi interface allows a single-threaded application to perform the same kinds of multiple, simultaneous transfers that multi-threaded programs can perform. It allows many of the benefits of multi-threaded transfers without the complexity of managing and synchronizing many threads.

To complicate matters somewhat more, there are even two versions of the multi interface. The event based one, also called multi_socket and the "normal one" designed for using with select(). See the libcurl-multi.3 man page for details on the multi_socket event based API, this description here is for the select() oriented one.

To use this interface, you are better off if you first understand the basics of how to use the easy interface. The multi interface is simply a way to make multiple transfers at the same time by adding up multiple easy handles into a "multi stack".

You create the easy handles you want, one for each concurrent transfer, and you set all the options just like you learned above, and then you create a multi handle with curl_multi_init and add all those easy handles to that multi handle with curl_multi_add_handle.

When you've added the handles you have for the moment (you can still add new ones at any time), you start the transfers by calling curl_multi_perform.

curl_multi_perform is asynchronous. It will only perform what can be done now and then return back control to your program. It is designed to never block. You need to keep calling the function until all transfers are completed.

The best usage of this interface is when you do a select() on all possible file descriptors or sockets to know when to call libcurl again. This also makes it easy for you to wait and respond to actions on your own application's sockets/handles. You figure out what to select() for by using curl_multi_fdset, that fills in a set of fd_set variables for you with the particular file descriptors libcurl uses for the moment.

When you then call select(), it'll return when one of the file handles signal action and you then call curl_multi_perform to allow libcurl to do what it wants to do. Take note that libcurl does also feature some time-out code so we advise you to never use very long timeouts on select() before you call curl_multi_perform again. curl_multi_timeout is provided to help you get a suitable timeout period.

Another precaution you should use: always call curl_multi_fdset immediately before the select() call since the current set of file descriptors may change in any curl function invoke.

If you want to stop the transfer of one of the easy handles in the stack, you can use curl_multi_remove_handle to remove individual easy handles. Remember that easy handles should be curl_easy_cleanuped.

When a transfer within the multi stack has finished, the counter of running transfers (as filled in by curl_multi_perform) will decrease. When the number reaches zero, all transfers are done.

curl_multi_info_read can be used to get information about completed transfers. It then returns the CURLcode for each easy transfer, to allow you to figure out success on each individual transfer.

SSL, Certificates and Other Tricks

[seeding, passwords, keys, certificates, ENGINE, ca certs]

Sharing Data Between Easy Handles

You can share some data between easy handles when the easy interface is used, and some data is share automatically when you use the multi interface.

When you add easy handles to a multi handle, these easy handles will automatically share a lot of the data that otherwise would be kept on a per-easy handle basis when the easy interface is used.

The DNS cache is shared between handles within a multi handle, making subsequent name resolving faster, and the connection pool that is kept to better allow persistent connections and connection re-use is also shared. If you're using the easy interface, you can still share these between specific easy handles by using the share interface, see libcurl-share.

Some things are never shared automatically, not within multi handles, like for example cookies so the only way to share that is with the share interface.

Footnotes

[1]

libcurl 7.10.3 and later have the ability to switch over to chunked Transfer-Encoding in cases where HTTP uploads are done with data of an unknown size.

[2]

This happens on Windows machines when libcurl is built and used as a DLL. However, you can still do this on Windows if you link with a static library.

[3]

The curl-config tool is generated at build-time (on Unix-like systems) and should be installed with the 'make install' or similar instruction that installs the library, header files, man pages etc.

[4]

This behavior was different in versions before 7.17.0, where strings had to remain valid past the end of the curl_easy_setopt call.

SEE ALSO

libcurl-errors, libcurl-multi, libcurl-easy