NAME

curl_multi_socket − reads/writes available data

SYNOPSIS

```
#include <curl/curl.h>

CURLMcode curl_multi_socket_action(CURLM * multi_handle,
                                   curl_socket_t sockfd, int ev_bitmask,
                                   int *running_handles);

CURLMcode curl_multi_socket(CURLM * multi_handle, curl_socket_t sockfd,
                            int *running_handles);

CURLMcode curl_multi_socket_all(CURLM *multi_handle,
                                int *running_handles);
```

DESCRIPTION

Alternative versions of *curl_multi_perform(3)* that allows the application to pass in the file descriptor/socket that has been detected to have "action" on it and let libcurl perform. This allows libcurl to not have to scan through all possible file descriptors to check for action. When the application has detected action on a socket handled by libcurl, it should call *curl_multi_socket_action(3)* with the **sockfd** argument set to the socket with the action. When the events on a socket are known, they can be passed as an events bitmask **ev_bitmask** by first setting **ev_bitmask** to 0, and then adding using bitwise OR (|) any combination of events to be choosen from CURL_CSELECT_IN, CURL_CSELECT_OUT or CURL_CSELECT_ERR. When the events on a socket are unknown, pass 0 instead, and libcurl will test the descriptor internally.

At return, the int **running_handles** points to will contain the number of still running easy handles within the multi handle. When this number reaches zero, all transfers are complete/done. Note that when you call *curl_multi_socket_action(3)* on a specific socket and the counter decreases by one, it DOES NOT necessarily mean that this exact socket/transfer is the one that completed. Use *curl_multi_info_read(3)* to figure out which easy handle that completed.

The curl_multi_socket functions inform the application about updates in the socket (file descriptor) status by doing none, one or multiple calls to the socket callback function set with the CURLMOPT_SOCKETFUNCTION option to *curl_multi_setopt(3)*. They update the status with changes since the previous time this function was called.

To force libcurl to (re-)check all its internal sockets and transfers instead of just a single one, you call **curl_multi_socket_all(3)**. This is typically done as the first function call before the application has any knowledge about what sockets libcurl uses.

Applications should call **curl_multi_timeout(3)** to figure out how long to wait for socket actions − at most − before doing the timeout action: call the **curl_multi_socket(3)** function with the **sockfd** argument set to CURL_SOCKET_TIMEOUT.

Usage of *curl_multi_socket(3)* is depricated, whereas the function is equivalent to *curl_multi_socket_action(3)*, when **ev_bitmask** is set to 0.

CALLBACK DETAILS

The socket **callback** function uses a prototype like this

```
int curl_socket_callback(CURL *easy,      /* easy handle */
                         curl_socket_t s, /* socket */
                         int action,      /* see values below */
```

```
void *userp,    /* private callback pointer */
void *socketp); /* private socket pointer */
```

The callback MUST return 0.

The *easy* argument is a pointer to the easy handle that deals with this particular socket. Note that a single handle may work with several sockets simultaneously.

The *s* argument is the actual socket value as you use it within your system.

The *action* argument to the callback has one of five values:

CURL_POLL_NONE (0)
: register, not interested in readiness (yet)

CURL_POLL_IN (1)
: register, interested in read readiness

CURL_POLL_OUT (2)
: register, interested in write readiness

CURL_POLL_INOUT (3)
: register, interested in both read and write readiness

CURL_POLL_REMOVE (4)
: deregister

The *socketp* argument is a private pointer you have previously set with *curl_multi_assign(3)* to be associated with the *s* socket. If no pointer has been set, socketp will be NULL. This argument is of course a service to applications that want to keep certain data or structs that are strictly associated to the given socket.

The *userp* argument is a private pointer you have previously set with *curl_multi_setopt(3)* and the CURLMOPT_SOCKETDATA option.

RETURN VALUE

CURLMcode type, general libcurl multi interface error code.

If you receive *CURLM_CALL_MULTI_PERFORM*, this basically means that you should call *curl_multi_socket(3)* again, before you wait for more actions on libcurl's sockets. You don't have to do it immediately, but the return code means that libcurl may have more data available to return or that there may be more data to send off before it is "satisfied".

NOTE that this only returns errors etc regarding the whole multi stack. There might still have occurred problems on individual transfers even when this function returns OK.

TYPICAL USAGE

1. Create a multi handle

2. Set the socket callback with CURLMOPT_SOCKETFUNCTION

3. Set the timeout callback with CURLMOPT_TIMERFUNCTION, to get to know what timeout value to use when waiting for socket activities.

4. Add easy handles

5. Call curl_multi_socket_all() first once

6. Provide some means to manage the sockets libcurl is using, so you can check them for activity. This can be done through your application code, or by way of an external library such as libevent or glib.

7. Wait for activity on any of libcurl's sockets, use the timeout value your calback has been told

8, When activity is detected, call curl_multi_socket_action() for the socket(s) that got action. If no activity is detected and the timeout expires, call *curl_multi_socket(3)* with *CURL_SOCKET_TIMEOUT*

9. Go back to step 7.

AVAILABILITY

This function was added in libcurl 7.15.4, although deemed stable since 7.16.0.

curl_multi_socket(3) is deprecated, use *curl_multi_socket_action(3)* instead!

SEE ALSO

curl_multi_cleanup(3), **curl_multi_init**(3), **curl_multi_fdset**(3), **curl_multi_info_read**(3), **the hiper-fifo.c example**